Andrew P. Black (Ed.)

LNCS 3586

ECOOP 2005 – Object-Oriented Programming

19th European Conference
Glasgow, UK, July 2005
Proceedings

ecoop
2005
European Conference
on Object-Oriented Programming

Springer

Andrew P. Black (Ed.)

ECOOP 2005 – Object-Oriented Programming

19th European Conference
Glasgow, UK, July 25-29, 2005
Proceedings

Springer

Volume Editor

Andrew P. Black
Portland State University
Maseeh College of Engineering and Computer Science
Department of Computer Science
Portland OR 97207, USA
E-mail: black@cs.pdx.edu

Library of Congress Control Number: 2005929057

CR Subject Classification (1998): D.1, D.2, D.3, F.3, C.2, K.4, J.1

ISSN	0302-9743
ISBN-10	3-540-27992-X Springer Berlin Heidelberg New York
ISBN-13	978-3-540-27992-1 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11531142 06/3142 5 4 3 2 1 0

Andrew P. Black (Ed.)

ECOOP 2005 - Object-Oriented Programming

19th European Conference
Glasgow, Scotland, UK, July 2005
Proceedings

ecoop
2005
European Conference
on Object-Oriented Programming

Springer

Volume Editor

Andrew P. Black
Department of Computer Science
Maseeh College of Engineering and Computer Science
Portland State University
Portland OR 97207, USA
E-mail: black@cs.pdx.edu

Preface

The 19th Annual Meeting of the European Conference on Object-Oriented Programming—ECOOP 2005—took place during the last week of July in Glasgow, Scotland, UK. This volume includes the refereed technical papers presented at the conference, and two invited papers. It is traditional to preface a volume of proceedings such as this with a note that emphasizes the importance of the conference in its respective field. Although such self-evaluations should always be taken with a large grain of salt, ECOOP is undisputedly the preeminent conference on object-orientation outside of the United States. In its turn, object-orientation is today's principal technology not only for programming, but also for design, analysis and specification of software systems. As a consequence, ECOOP has expanded far beyond its roots in programming to encompass all of these areas of research—which is why ECOOP has remained such an interesting conference.

But ECOOP is *more* than an interesting conference. It is the nucleus of a technical and academic *community*, a community whose goals are the creation and dissemination of new knowledge. Chance meetings at ECOOP have helped to spawn collaborations that span the boundaries of our many subdisciplines, bring together researchers and practitioners, cross cultures, and reach from one side of the world to the other. The ubiquity of fast electronic communication has made maintaining these collaborations easier than we would have believed possible only a dozen years ago. But the role of conferences like ECOOP in *establishing* collaborations has not diminished. Indeed, as governments make it harder to travel and emphasize the divisions between nations, it becomes ever more important that we strengthen our personal and professional networks and build bonds between individuals, institutions, and countries.

As we have moved into the electronic age, we have realized that it is not so much a shared locality or a shared language that defines a community, but a shared set of values. As a scientific community, we value evidence-based inquiry. We value truth, and demand a high standard of evidence in ourselves and in our colleagues as we search for it. We value integrity, because without integrity we cannot build upon the results of our colleagues, and the scientific process will grind to a halt. We value technology, because technology gives us new tools in the search for truth. We value clear communication, because without communication new knowledge cannot be evaluated by our colleagues, or influence the way that others think. We value the slow social process by which one individual's discoveries become accepted knowledge, because we recognize that this process is the best way yet found to minimize subjectivity.

ECOOP's refereed technical program consisted of 24 papers that were selected by the Program Committee from 172 submissions. Every paper was read by at least three members of the Program Committee; some papers, which appeared controversial, were read by four or five PC members. In addition, the committee sought the opinions of 147 co-reviewers, selected because of their expertise on particular topics. As is usual, the selection took place at a two-day meeting. However, because of the difficulty that many of the European PC members would have experienced in traveling to my home institution in the United States, the meeting was held in Bern, Switzerland. I am very grateful to Prof. Oscar Nierstrasz, a founding member of AITO and Programme Chair for ECOOP '93, for offering the use of his facilities at the SCG in Bern, and to Oscar, Therese Schmid and the students of the SCG who went to extraordinary lengths to help me with the local arrangements.

For many of us, the invited talks, tutorials and workshops at ECOOP are at least as important as the talks based on refereed papers. This year, we featured two invited technical talks, described more fully on page VII, in addition to the banquet address, which was given by Dr. Gilad Bracha. The schedule for the week included 19 workshops and 16 tutorials, selected for their topicality, interest and diversity. Many of the tutorials were offered more than once, to reduce scheduling conflicts for attendees who wished to attend multiple events. It also gave me great pleasure to host Prof. Emeritus Peter Wegner as guest of honor at the conference banquet. Peter has the distinction of having defined the term "object-oriented language" in his 1987 OOPSLA paper, and has been involved with ECOOP as a workshop participant and panelist at least since 1988. While traveling to Lisbon to give the banquet address at ECOOP '99, Peter was struck down by a bus in London, and suffered life-threatening injuries. He has made a most remarkable recovery, and I was absolutely delighted to be able to welcome him back into the ECOOP community.

The continued success of ECOOP depends on the dedication and hard work of a large number of people; not only is most of this work performed voluntarily, but we compete with each other to volunteer! In addition to me, 24 distinguished researchers served on the Program Committee, writing sometimes lengthy reviews of many papers, working very long hours to meet the conference deadlines, and behaving (almost always!) in a most professional manner. The conference could not have taken place at all without the efforts of the 373 authors who submitted papers, the board of AITO, which sponsored the conference, the conference Organizing Committee, the Tutorials and Workshops Committees, and the local organizers and student volunteers. Richard van de Stadt also deserves a special mention for the excellence of his technical support through CyberChairPRO.

June 2005

Andrew P. Black
ECOOP 2005 Program Chair

The AITO Dahl-Nygaard Prize

It was a great loss to our community when both Ole-Johan Dahl and Kristen Nygaard passed away in 2002, not long after ECOOP in Málaga. Pioneers in the areas of programming and simulation, their foundational work on object-oriented programming, made concrete in the Simula language, can now be seen as one of the most significant inventions in software engineering. Their key ideas took shape around 1965, but more than 20 years were to pass before these ideas were fully absorbed into and appreciated by the broader software community. Since then, object-orientation has profoundly transformed the landscape of software design and the process of software development.

In remembrance of Dahl and Nygaard's scholarship and their enthusiastic encouragement of young researchers, AITO has established a pair of annual prizes. The senior prize is awarded to a researcher with outstanding career contributions, and the junior prize is awarded to a younger researcher who has demonstrated great potential for following in the footsteps of these pioneers.

This year, the first time that the prizes have been awarded, the Prize Committee selected Bertrand Meyer to receive the Dahl-Nygaard Senior Prize, and Gail Murphy to receive the Dahl-Nygaard Junior Prize.

Bertrand Meyer was one of the most influential researchers in the 1980s, in the initial period of wide adoption of object-oriented programming. He designed the Eiffel language, which pioneered the concept of *design by contract.* He provided strong arguments for object-oriented software architecture in his book "Object-Oriented Software Construction", which remains to this day a highly influential work. Many of his contributions have proven to be of lasting value.

Like Nygaard, Meyer has not backed away from controversy and has consistently followed his own vision of object orientation. Design by contract established an essential bridge between axiomatic specification and object-oriented programming. Bertrand Meyer is currently Professor of Software Engineering at ETH Zürich in Switzerland. His research on trusted components continues to explore challenging problems in software engineering.

Gail Murphy has shown promising potential as a young researcher by proposing innovative ideas and by proving that these are conceptually sound and realistically implementable. She focuses her research and teaching on software engineering, and she has made contributions to understanding and reducing the problems associated with the evolution of large software systems.

Gail Murphy is currently an Associate Professor at the University of British Columbia in Canada. Like Dahl and Nygaard, Murphy challenges students to examine new proposals with a disciplined and questioning eye. She is preparing a new generation of researchers by encouraging the development of sound theories backed by solid prototype implementations.

Both Meyer and Murphy agreed to present lectures at ECOOP 2005. Invited papers corresponding to these lectures are included in this volume. Meyer's paper is entitled "Attached Types and Their Application to Three Open Problems of Object-Oriented Programming" and begins on page 1. Murphy's paper is entitled "The Emergent Structure of Development Tasks" and begins on page 33.

June 2005

Jean Bézivin
Markku Sakkinen
Dave Thomas
AITO Dahl-Nygaard Prize Committee

Organization

ECOOP 2005 was organized by the Universities of Glasgow and Strathclyde, under the auspices of AITO (Association Internationale pour les Technologies Objets), and in cooperation with ACM SIGPLAN.

Executive Committee

General Co-chairs
- Peter Dickman (University of Glasgow)
- Paddy Nixon (University of Strathclyde)

Program Chair
- Andrew P. Black (Portland State University)

Organizing Chair
- Peter Dickman (University of Glasgow)

Organizing Committee

Workshops
- Marc Roper (University of Strathclyde)
- Wolfgang De Meuter (Vrije Universiteit Brussels)

Tutorials
- Karen Renaud (University of Glasgow)
- Elisa Baniassad (The Chinese University of Hong Kong)

Demos and Posters
- Rob Pooley (Heriott-Watt University)

PhD Workshop/Doctoral Symposium
- Alex Potanin (Victoria University of Wellington)
- David Lievens (University of Strathclyde)

Treasurer
- David Watt (University of Glasgow)

Catering Liaison/Coordination
- Simon Gay (University of Glasgow)

Venue Liaison/Coordination
- Sotirios Terzis (University of Strathclyde)

IT Facilities Liaison
- Tony Printezis (Sun Microsystems Laboratories)

Registration
  Murray Wood (University of Strathclyde)
Sponsorship
  Steve Neely (University of Strathclyde)
Publicity and Web
  Ian Ferguson (University of Strathclyde)
  Richard Cooper (University of Glasgow)
  Karen Renaud (University of Glasgow)
Workshops Review Committee
  Marc Roper (University of Strathclyde)
  Wolfgang De Meuter (Vrije Universiteit Brussels)
  Martine Devos (Avaya Labs Research)
  Michel Tilman (Real Software)
  Maximo Prieto (Universidad de la Plata)
  Stéphane Ducasse (University of Bern)
  Ralf Lämmel (CWI Amsterdam)
Doctoral Symposium Committee
  Alex Potanin (Victoria University of Wellington)
  David Lievens (University of Strathclyde)
  István Zólyomi (Eötvös Loránd University)
  Gregory de Fombelle (Thales Research and UPMC (Paris 6))
  Jérôme Darbon (Ecole Nationale Supérieure des Télécommunications)

As Organizing Chair I would like to thank the support staff and event planners at the Glasgow Convention Bureau, the Scottish Exhibition and Conference Centre, the Moat House Hotel Glasgow, the Glasgow Science Centre and the Arches. Administrative and support staff at the Universities of Glasgow and Strathclyde also contributed to the organization of ECOOP 2005 and their efforts are much appreciated. Most importantly, thanks are due to the many student volunteers who kept things running smoothly during the meeting and who were critical to the success of the conference.

Peter Dickman

Program Committee

Mehmet Akşit	University of Twente, Netherlands
Luca Cardelli	Microsoft, UK
Shigeru Chiba	Tokyo Institute of Technology, Japan
Yvonne Coady	University of Victoria, Canada
William Cook	University of Texas, Austin, USA
Theo D'Hondt	Vrije Universiteit Brussel, Belgium
Christophe Dony	Montpellier-II University, France
Stphane Ducasse	University of Bern, Switzerland
Erik Ernst	University of Aarhus, Denmark
Richard P. Gabriel	Sun Microsystems, USA
Tony Hosking	Purdue University, USA
Jean-Marc Jzquel	Irisa/Univ. Rennes 1, France
Eric Jul	University of Copenhagen, Denmark
Luigi Liquori	INRIA, France
James Noble	Victoria University of Wellington, New Zealand
Martin Odersky	EPFL, Switzerland
Christian Queinnec	Universit Pierre et Marie Curie, France
Martin Robillard	McGill University, Canada
Jrg Striegnitz	Research Centre Jlich, Germany
Peri Tarr	IBM Research, USA
Dave Thomas	Bedarra, Canada
Mads Torgersen	University of Aarhus, Denmark
Todd Veldhuizen	Chalmers University, Sweden
Allen Wirfs-Brock	Microsoft, USA

Referees

Jonathan Aldrich
Davide Ancona
Francoise André
Gabriela Arévalo
Uwe Assmann
Thomas Baar
Arnaud Bailly
Jennifer Baldwin
Daniel Bardou
Mike Barnett
Don Batory
Benoit Baudry
Klaas van den Berg
Alexandre Bergel
Lodewijk Bergmans
Gavin Bierman
Stephen Blackburn
Mireille Blay-Fornarino
Bard Bloom
Guillaume Bonfante
Chandrasekhar Boyapati
Gilad Bracha
Jean-Pierre Briot
Pim van den Broek
Dennis Brylow
Michele Bugliesi
Cristiano Calcagno
Michelle Cart
Emmanuel Chailloux
Stephen Chong

Mark C. Chu-Carroll
Dave Clarke
Thomas Cleenewerck
Pierre Cointe
Pascal Costanza
Vincent Cremet
Tom Van Cutsem
Ferruccio Damiani
Jessie Dedecker
Robert DeLine
Marcus Denker
Iulian Dragos
Sophia Drossopoulou
Roland Ducournau
Pascal Durr
Peter Ebraert
Tatjana Eitrich
Burak Emir
Jean Ferrié
John Field
Franck Fleurey
Remi Forax
Robert Fuhrer
Markus Gälli
Vladimir Gapeyev
Jacques Garrigue
Sofie Goderis
Paul Grace
Orla Greevy
Wolfgang Grieskamp
Dan Grossmann
Gurcan Gulesir
Mads Haahr
Richard Hamlet
William Harrison
Tom Hirschovitz
Marianne Huchard
Atsushi Igarashi
Radha Jagadeesan
Suresh Jagannathan
Nadeem Jamali
Harmen Kastenberg
Andy Kellens
Pertti Kellomäki
Andrew Kennedy
Joseph Kiniry
Shriram Krishnamurthi
Neel Krishnaswami
Doug Lea
Christopher League
Gary T. Leavens
Thérèse Libourel
Henry Lieberman
Hanbing Liu
Cristina Videira Lopes
Roberto Lopez-Herrejon
Jacques Malenfant
Jean-Yves Marion
Michel Mauny
Wolfgang De Meuter
Isabel Michiels
Nikolay Mihaylov
Todd Millstein
Stijn Mostinckx
Peter Müller
Gail Murphy
Istvan Nagy
Srinivas Nedunuri
Philippe Nguyen
Joost Noppen
Johan Nordlander
Nathaniel Nystrom
Harold Ossher
Ellen Van Paesschen
Amit Paradkar
Didier Parigot
Renaud Pawlak
Frédéric Peschanski
Filip Pizlo
Noël Plouzeau
Erik Poll
Laura Ponisio
Anne Pons
Isabelle Puaut
Philip Quintslund
Vadakkedathu T. Rajan
Arend Rensink
Coen De Roover
Guido van Rossum
Claudio Russo

Barbara Ryder
Amr Sabry
Lionel Seinturier
Bernard Serpette
Manuel Serrano
Jesper Honig Spring
Nathanael Schärli
Jim Steel
Gerson Sunyé
Bedir Tekinerdoğan
Hendrik Tews
Peter Thiemann
Robert Tolksdorf
Mads Torgersen
Yves Le Traon
Christelle Urtado
Sylvain Vauttier
Mirko Viroli
Jan Vitek
John Vlissides
Adam Welc
Ben Wiedermann
Rebecca Wirfs-Brock
Tobias Wrigstad
Roel Wuyts
Matthias Zenger
Tewfik Ziadi

Table of Contents

Attached Types and Their Application to Three Open Problems of Object-Oriented Programming

Bertrand Meyer

ETH Zurich and Eiffel Software
`http://se.inf.ethz.ch – http://www.eiffel.com`

Abstract. The three problems of the title — the first two widely discussed in the literature, the third less well known but just as important for further development of object technology — are:

- Eradicating the risk of **void calls**: *x*.*f* with, at run time, the target *x* not denoting any object, leading to an exception and usually a crash.
- Eradicating the risk of "**catcalls**": erroneous run-time situations, almost inevitably leading to crashes, resulting from the use of covariant argument typing.
- Providing a simple way, in concurrent object-oriented programming, to **lock** an object handled by a remote processor or thread of control, or to access it *without* locking it, as needed by the context and in a safe way.

A language mechanism provides a combined solution to all three issues.

This mechanism also allows new solutions to two known problems: how to check that a certain object has a certain type, and then use it accordingly ("Run-Time Type Identification" or "downcasting"), for which it may provide a small improvement over previously proposed techniques; and how to provide a "once per object" facility, permitting just-in-time evaluation of certain object properties.

The solution relies on a small extension to the type system involving a single symbol, the question mark. The idea is to declare certain types as "attached" (not permitting void values), enforce some new validity rules that rule out void calls, and validate a number of common programming schemes as "Certified Attachment Patterns" guaranteed to rule out void calls. (In addition, the design replaced an existing type-querying construct by a simpler one.)

The mechanism is completely static: all checks can be performed by compilers as part of normal type system enforcement. It places no undue burden on these compilers — in particular, does not require dataflow analysis — and can be fairly quickly explained to programmers. Existing code, if reasonably well-written, will usually continue to work without change; for exceptions to this rule, often reflecting real risks of run-time crashes, backward-compatible options and a clear transition path are available.

The result is part of the draft ECMA (future ISO) standard for Eiffel.

There is one and only one kind of acceptable language extension: the one that dawns on you with the sudden self-evidence of morning mist. It must provide a complete solution to a real problem, but usually that is not enough: almost all good extensions solve several potential prob-

lems at once, through a simple addition. It must be simple, elegant, explainable to any competent user of the language in a minute or two. (If it takes three, forget it.) It must fit perfectly within the spirit and letter of the rest of the language. It must not have any dark sides or raise any unanswerable questions. And because software engineering is engineering, and unimplemented ideas are worth little more than the whiteboard marker with serves to sketch them, you must see the implementation technique. The implementors' group in the corner of the room is grumbling, of course — how good would a nongrumbling implementor be? — but you and they see that they can do it.

When this happens, then there is only one thing to do: go home and forget about it all until the next morning. For in most cases it will be a false alarm. If it still looks good after a whole night, then the current month may not altogether have been lost.

From "Notes on Language Design and Evolution" in **[6]**

1 Overview

The design of a programming language is largely, if the designer cares at all about reliability of the resulting programs, the design of a type system. Object-oriented programming as a whole rests on a certain view of typing, the theory of abstract data types; this makes it natural, when searching for solutions to remaining open problems, to turn for help to typing mechanisms.

One such problem is **void-safety**: how to guarantee that in the fundamental object-oriented operation, a feature call *x*.*f* (*args*), the target *x* will always, at execution time, denote an object. If it does not — because *x* is "void" — an exception will occur, often leading to a crash.

This article shows that by fine-tuning the type system we may remove this last significant source of run-time errors in object-oriented programs. The basic language extension is just one symbol (the question mark).

As language extensions should, the mechanism yields other benefits beyond its initial purpose. It provides a solution to the "catcall" issue arising from covariant argument redefinition; a better technique of run-time object type identification; a flexible approach to object locking in concurrent programming; and a simple way to perform lazy computation of attributes ("once per object").

1.1 Mechanism Summary

Here is a capsule description of the mechanism:

- A call *x*.*f* (*args*) is only valid (as enforced statically) if its target *x* is **attached**.
- The simplest way for a variable to be attached is to be declared of an "attached type", guaranteeing that no value can be void. Types are indeed attached by default. To get the "detachable" version of a type *T* (permitting void values) use **?** *T*.

- For variables of a detachable type, some simple and common program schemes guarantee a non-void value. For example, immediately after **create** *x*... or the test **if** *x* /= *Void* **then** ... , *x* is not void. The mechanism uses a small catalog of such "*Certified Attachment Patterns*" or CAPs, easy for programmers to understand and for compilers to implement. A variable used in such a CAP is attached (again, statically), even if its type is not. CAPs are particularly important in ensuring that reasonably-written existing software will run unmodified. Initial evaluation suggests that this will be the case with the vast majority of current code.
- Outside of these patterns, a call of target *x* requires *x* to be of an attached type. The remaining problem is to guarantee that such a variable will never be void. The basic rule concerns assignment and argument passing: if the target is attached, the source must be attached too. This leaves only the question of initialization: how to ensure that any attached variable is, on first access, not void.
- Some types guarantee non-void initialization by providing a default initialization procedure that will produce an object. We call them "self-initializing types".
- A variable that is not of such a type may provide its own specific initialization mechanism. We call it a "self-initializing variable".
- Generic classes can use a question mark to specify a self-initializing type parameter.
- This leaves only the case of a variable that could be accessed while void (because no CAP applies, the type is not self-initializing, and neither is the variable itself). The "Object Test" construct makes it possible to find out if the variable is attached to an object of a specific type, and then to use it safely.

1.2 The Void Safety Issue

The basic idea of typed object-oriented languages is to ensure, thanks to validity rules on program texts enforced statically (at compile-time), that the typical object-oriented operation, *x*. *f* (*args*), known as a "qualified call", will never find *x* attached to an object not equipped to execute the operation *f*. The validity rules essentially require the programmer to declare every variable, routine argument, routine result and other entity with an explicit type (based on a class, which must include the appropriate *f* with the appropriate arguments), and to restrict polymorphic assignments *x* := *y*, as well as actual-to-formal argument associations, to those in which the type of *y* **conforms** to the type of *x*; conformance is governed by inheritance between classes, so that if *f* is available for the type of *y* it will also be available, with a compatible signature, for the type of *x*.

This technique, pioneered by Eiffel and Trellis-Owl and since implemented in various ways in typed O-O languages, eliminates many potential run-time errors, and has succeeded in establishing static typing firmly. But — notice the double negation in the above phrasing of the "basic idea" — it only works if the target entity, *x*, is **attached** to an object at the time of execution. Rather than directly denoting an object, *x* is often a *reference* to a potential object; to support the description of flexible data structures, programming languages generally permit a reference to be *void*, or "null", that is to say attached to no object. If *x* is void at the time of the call, an exception will result, often leading to a crash.

The initial goal for the work reported here was once and for all to remove this sword of Damocles hanging over the execution of even a fully type-checked program.

1.3 General Description

The basis of the solution is to extend the type system by defining every type as "*attached*" or "*detachable*", where an attached type guarantees that the corresponding values are never void. Attached is the default. A qualified call, *x*.*f* (*args*), is now valid **only** if the type of *x* is attached. Another new validity rule now allows us to assign (or perform argument passing) from the attached version of a type to the detachable version, but not the other way around without a check of non-voidness. Such a check, applied to an expression *exp* of a detachable type, is a new kind of boolean expression: an "Object Test" of the form {*x*: *T*} *exp*, where *T* is the desired attached type and *x* is a fresh variable. In the Conditional instruction

```
if {x: T} exp then                                                    /1/
        ... Instructions, in particular calls of the form x.f (args)...
end
```

if the Object Test evaluates to true, meaning that *exp* is indeed attached to an object of type *T*, *x* is bound to that value of *exp* over the "scope" of the Object Test, here the whole **then** clause. Calls of target *x* are then guaranteed to apply to a non-void target over that scope. It is necessary to use such a locally bound variable, rather than directly working on *exp*, because if *exp* is a complex expression or even just an attribute of the class many kinds of operation occurring within the **then** clause, such as calls to other routines of the class, could perform assignments that make *exp* void a gain and hence hang the sword of Damocles back up again. The variable *x* is a "read-only", like a formal routine argument in Eiffel: it cannot figure as the target of an assignment, and hence will keep, over the scope of the Object Test, the original value of *exp*, guaranteed to be non-void.

The Object Test resembles mechanisms found in typed object-oriented languages under names such as "Run-Time Type Identification", "type narrowing", "downcasting", and the "with" instruction of Oberon; it addresses their common goal in a compact and general form and is intended to subsume them all. In particular, it replaces Eiffel's original "Assignment Attempt" instruction, one of the first such mechanisms, written *x* ?= *exp* with (in the absence of a specific provision for attached types) the semantics of assigning *exp* to *x* if *exp* happens to be attached to an object of the same type as *x* or conforming, and making *x* void otherwise. An assignment attempt is typically followed by an instruction that tests *x* against *Void*. The Object Test, thanks to its bound variable and its notion of scope, merges the assignment and the test.

Relying on the Object Test instruction alone would yield a complete solution of the Void Call Eradication problem, but would cause considerable changes to existing code. Sometimes it is indeed necessary to add an Object Test for safety, but in a huge number of practical cases it would do nothing but obscure the program text, as the context guarantees a non-void value. For example, immediately after a creation instruction **create Result**..., we know that **Result**, even if declared of a detachable type, has an attached value and hence can be used as the result of a function itself declared attached. We certainly do not want in such a case to be forced to protect **Result** through an Object Test, which would be just noise. An important part of the mechanism is the notion of *Certified Attachment Patterns*: a catalog of program schemes officially guaranteeing that a certain variable, even if declared of an attached type, will in certain contexts always have a certifiably attached value. The catalog is limited to cases that can be safely and universally guaranteed correct, both easily explainable to programmers and easily implementable by compilers; these cases cover a vast number of practical situations, ensuring that the Object Test, however fundamental to the soundness of the approach as a whole, remains — as it should be — a specialized technique to be used only rarely.

An immediate consequence of these techniques will be to remove preconditions, occurring widely in libraries of reusable classes as well as in production applications, of the form **require** *x* / = *Void* for a routine argument *x* (sometimes for an attribute as well). Informal surveys shows that in well-written Eiffel code up to 80% of routines contain such a precondition. With the new type system, it is no longer necessary if we declare *x* of an attached type. Going from preconditions to a static declaration, and hence a compile-time check, is a great boost to reliability and a significant simplification of the program text.

To go from these basic ideas to a full-fledged language mechanism that delivers on the promise of total, statically-enforced Void Call Eradication, the solution must address some delicate issues:

- In a language framework guaranteeing for reliability and security that all variables, in particular object fields, local variables of routines and results of functions, are automatically initialized (an idea also pioneered by Eiffel and widely adopted by recent languages), how to ensure that variables declared of an attached type are indeed initialized to attached values.
- How to handle attached type in the context of genericity. For example, the Eiffel library class *ARRAY* [*G*] is generic, describing arrays of an arbitrary type *G*. Sometimes the corresponding actual parameter will be attached, requiring — or not! — automatic initialization of array entries; sometimes it will be detachable, requiring automatic initialization of all entries to *Void*. It would be really unpleasant, for this and all other container classes, to have to provide two versions, one for detachable types and one for attached types, or even three depending on initialization requirements for attached types. The solution to this issue is remarkably simple (much shorter to explain than the details of the issue itself): if a generic class needs to rely on automatic initialization of variables of the formal generic type (here *G*), make this part of the declaration for the parameter, requiring clients to provide an initialization mechanism for actual parameters that are attached types.
- How to make the whole mechanism as invisible as possible to programmers using the language. We must not force them to use any complicated scheme to attain ordinary results; and we must guarantee an "effect of least surprise". In other words they should be able to write their application classes in a simple and intuitive way, the way they have always done, even if they do not understand all the subtleties of attachment, and it is then our responsibility to ensure that they get safely operating programs and the semantics corresponding to their intuition.

- How to ensure that the resulting type system achieves its goal of total Void Call Eradication. The authors of Spec#, a previous design which influenced this work, write that they expect "*fewer unexpected non-null reference exceptions*" **[3]**. We are more ambitious and expect to remove such exceptions entirely and forever. Here it must be mentioned that although we believe that the design described here reaches this goal we have not provided a mathematical proof or, for that matter, do not yet have a formal framework in which to present such a proof.
- In the case of Eiffel, a well-established language with millions of lines of production code, how to provide a smooth transition to the new framework. The designers of Spec# have the advantage of working on a new research language; Eiffel has commercial implementations with heavy customer investment in business-critical applications, and we must guarantee either backward compatibility or a clear migration path. This alone is a make-or-break requirement for any proposed Eiffel solution.

Our solutions to these issues will be described below.

In finalizing the mechanism we realized that it appears to help with two other pending issues, one widely discussed and the other more esoteric at first sight but important for the future of object technology:

- A *covariant* type system (where both arguments and results of functions can be redefined in descendant classes to types conforming to their originals) raises, in a framework supporting polymorphism and dynamic binding, the specter of run-time type mismatches, or "catcalls", another source of crashes. We suggest the following solution to remove this other threat to the reliability of our software: permit covariant redefinition of an argument (covariant result types are not a problem) *only* if the new type is detachable. Then the new version must perform an Object Test, and no catcall will result. This is a way of allowing the programmer to perform covariant redefinition but forcing him to recognize that polymorphism may yield at run time an actual argument of the old type, and to deal with that situation explicitly. The rule also applies to the case of "anchored types", which is a form of implicit covariance, and appears to resolve the issue.
- An analysis of what it takes to bring *concurrent programming* to the level of quality and trust achieved by sequential programming, and bring it up to a comparable level of abstraction, has led to the development of the SCOOP mechanism [11] based on the transposition to a concurrent context of the basic ideas of Design by Contract. One of the conclusions is to allow a call *x*.*f* (*args*) to use a target *x* representing a "separate" object — an object handled by a different processor — and hence to support asynchronous handling, one of the principal benefits of concurrency, *only* if *x* is one of the formal arguments of the enclosing routine. Then a call to that routine, using as actual argument for *x* a reference to such a separate object, will block until the object becomes available, and then will place an exclusive hold on it for the duration of the routine's execution. But it turns out that, conversely, a call using a separate actual argument should not always reserve the object; for example we might only want to pass to another routine a reference to that object, without performing any call on it. It would not be appropriate to decide on

the basis of the routine's code whether object reservation is needed or not, as a kind of compiler optimization: clients should not have to know that code, and in any case the body of a routine may be redefined along the inheritance hierarchy, so that the language would not guarantee a specific semantics for a routine under polymorphism. Instead, the rules will now specify that passing a separate object as actual argument causes the call to place a reservation on the object *if and only if* the corresponding formal argument is declared of an *attached* type. If not, the routine can assign the argument to another variable, or pass it on to another routine; the target of the assignment, or the corresponding formal argument, must themselves be of an unattached type in accordance with the basic rule stated above. To perform a *call* using such an argument as target, one must check its attachment status, relying as usual on an Object Test; the final new semantic rule is that an Object Test on a separate expression will (like its use as actual argument to a routine with a corresponding attached formal) cause reservation of the object. So a simple convention to define the effect of combining two type annotations, "separate" and "attached", appears to provide the flexible and general solution sought.

In passing, we will see that the mechanism additionally addresses two problems for which solutions were available before, but perhaps addresses them better. One of the problem is Run-Time Type Identification: the Object Test construct provides a simple and general approach to this issue. The other, for which Eiffel already provided a specific mechanism, is "once per object": how to equip a class with a feature that will be computed only once for a given object, and only if needed at execution time. For example a field in objects representing the stock of a company might denote the price history of the share over several years. If needed, this field, pointing to a large list of values, will have to be initialized from a database. If only because of the time and space cost, we want to retrieve these values only if needed, and then the first time it is needed.

The following sections detail the mechanism and these applications.

2 Previous Work, Context and Acknowledgments

The "non-null types" of Spec# are the obvious inspiration for the design presented here. It is a pleasure to acknowledge the influence of that work. Our goal has been to try for a simpler and more general mechanism. The reader who would like to compare the two designs should note that references to Spec# in this article are based on 2003-2004 publications **[3]** **[1]** and check more recent work since Spec# has been progressing rapidly.

Other work addressing some of the same issues has included the Self language's attempt to eliminate Void values altogether **[2]** and my own earlier (too complicated) attempt to provide void-avoidance analysis **[10]**. I also benefited from early exposure to the type system work of Erik Meijer and Wolfram Schulte **[5]**.

The design reported here resulted from the work of the ECMA standardization effort for Eiffel (ECMA TC39-TG4), intended to yield an ISO standard **[4]**. The basic ideas are due to Éric Bezault, Mark Howard, Emmanuel Stapf (TG4 convener and secretary) and Kim Waldén. Mark Howard first proposed, I believe, the idea of replacing Eiffel's Assignment Attempt by a construct also addressing void call eradication. The actual design of that construct, the Object Test, is due to Karine Arnout

and Éric Bezault. This article largely reports on the ideas developed by this group of people. As the editor of the standard I bear responsibility for any remaining mistakes in the mechanism and of course in this article.

Numerous discussions with Peter Müller from ETH have been particularly fruitful in shaping the ideas. The application of the mechanism to SCOOP (the last problem) is part of joint work with Piotr Nienaltowski of ETH. Also helpful have been comments on the Eiffel draft standard from David Hollenberg and Paul-Georges Crismer.

In addition I am grateful to Andrew Black and Richard van de Stadt for their tolerance and kind assistance (extending beyond the normal duties of editors) in getting this article to press.

3 Syntax Extension

In Eiffel's spirit of simplicity the advances reported here essentially rely on one single-letter symbol: it is now permitted to prefix a type by a question mark, as in

```
x: ? T
```

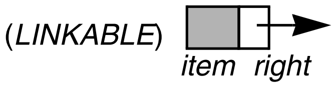
instead of the usual *x*: *T*. (The other syntactical novelty, Object Test, is not an addition but a replacement for the previous Assignment Attempt mechanism.) The question mark turns the type from attached to detachable. It is also possible to prefix a formal generic parameter with a question mark, as in

```
class ARRAY [? G] ...
```

with semantics explained in section 7.

The standards committee decided that in the absence of a question mark **types are attached by default** and hence do not support *Void* as a possible value. This is based on the analysis that void values are of interest to authors of fundamental data structure libraries such as EiffelBase **[7]**, which include classes representing linked data structures such as void-terminated linked lists, but much less to authors of application programs; classes *COMPANY_STOCK* in a financial application or *LANDING_ROUTE* in an aeronautic application are unlikely to require support for void values. So we ask professional library developers working on the basic "plumbing" to specify the possibility of void values when they need it, by using detachable types for example in the declaration of the neighboring item in class *LINKABLE* [*G*] describing linked list items:

```
right: ? LINKABLE [G]
```



but leave application programmers in peace when, as should usually be the case, they don't care about void values and, more importantly, don't want to worry about the resulting possibility of void calls.

This choice of default semantics raises a backward compatibility problem in the context, mentioned above, of preserving the huge commercial investment of Eiffel users; in the previous versions of the language, reference types support void by default, and some programs take advantage of that convention. To address this issue, we provide the symbol **!** as a transition facility. **!** *T* means the attached version of type *T*. In standard Eiffel this will mean the same as *T*, so the exclamation mark symbol is redundant. But offering an explicit symbol enables compilers to provide a migration option whereby the default semantics is reversed (*T* means **?** *T*), compatible with the previous convention. Programmers can then continue to use their existing classes with their original semantics, while starting to take advantage of void-call avoidance guarantees by declaring attached types with the explicit **!**. In the final state, the need for **!** will go away. In the rest of this article we stick with the Standard option: we don't need to use **!** at all, with the understanding that *T* means **!** *T*.

The **?** and **!** symbols are inspired by the conventions of Spec#. There has been criticism on the part of some Eiffel users that these are cryptic symbols ("*it looks like C++ !*") not in the Eiffel style; the symbol **!** in particular has bad karma since it was part of a short-lived syntax variant for the creation instruction now written in the normal Eiffel style as **create** *x*. Although the symbols have the benefit of brevity, they might similarly go away in favor of keywords, not affecting the validity rules, semantics and discussion of the present article.

To understand the rest of that discussion, note that Eiffel has two kinds of type: *reference* types, the default, whose values are reference to objects (or void in the case of detachable types); and *expanded* types, equipped with copy semantics. (The "value" types of C# are a slightly more restricted form of expanded types.) A type is expanded if it is based on a class declared as **expanded class** *C* ... rather than just **class** *C* ... Expanded types serve in particular to represent subobject fields of objects, as well as to model the basic types such as *INTEGER* and *REAL*, enabling Eiffel to have a consistent type system entirely based on the notion of class. Obviously expanded types do not support *Void* as one of their possible values. In the rest of this discussion the term "attached type" covers both non-detachable reference types (the most common case) and expanded types; that is to say, every type except a (reference) detachable type declared explicitly as **?** *T*.

4 Constraints on Calls and Attachment

The fundamental new constraint ensuring avoidance of void calls restricts the target of a qualified call:

Target Validity rule

A qualified call *a*.*f* or *a*.*f* (*args*) is valid only if the target expression *a* is attached.

An expression *a* is said to be attached, in the usual case, if its type is attached. This notion will be slightly generalized below.

A general note on the style of language description: "validity rules" in the specification of Eiffel **[6] [4] [12]** stand between syntax and semantics; they supplement the syntax by placing constraints (sometimes known as "static semantics") on acceptable language elements. Unlike in many other language descriptions, Eiffel's validity rules are always phrased in "if and only if" style: they don't just list individual permitted and prohibited cases, but give an exhaustive list of the necessary *and sufficient* conditions for a construct specimen to be valid, thus reinforcing programmer's confidence in the language. This property obviously does not apply to the rules as given in this article, since it is not a complete language description. The Target Validity rule, for example, appears above in "only if" style since it supplements other clauses on valid calls (such as *a* being of a type that has a feature *f* with the appropriate arguments, exported to the given client). The rules respect the spirit of the language definition, however, by essentially specifying all the supplementary clauses added to the existing rules.

The Target Validity rule will clearly ensure eradication of void calls if attached types live up to their name by not permitting void values at run time; the discussion will now focus on how to meet this requirement.

The other principal new constraint on an existing construct governs attachment. The term "attachment", for source *y* and target *x*, covers two operations: the assignment *x* := *y*, and argument passing *f* (..., *y*, ...) or *a*.*f* (..., *y*, ...) where the corresponding formal argument in *f* is *x*. The basic existing rule on attachment is *conformance* or *convertibility* of the source to the target; conformance, as mentioned, is based on inheritance (with provision for generic parameters), and convertibility is based on the Eiffel mechanism, generalizing ordinary conversions between basic types such as *INTEGER* and *REAL*, and allowing programmers to specify conversions as part of a class definition. Now we add a condition:

Attachment Consistency rule

An attachment of source *y* and target *x*, where the type of *x* is attached, is permitted only if the type of *y* is also attached.

This rule is trivially satisfied for expanded types (the only type that conforms to an expanded type *ET* is *ET* itself) but new for attached reference types.

A companion rule lets us, in the redefinition of a feature in a descendant of the original class, change a result type from detachable to attached, and an argument type from attached to detachable. The rationale is the same, understood in the context of polymorphism and dynamic binding.

This rule narrows down the risk of void call by guaranteeing that if a void value arises somewhere it will not be transmitted, through assignment or argument passing, to variables of attached types. There remains to guarantee that the values *initially* set for targets of attached type can never be void. This sometimes delicate initialization issue will indeed occupy most of the remaining discussion.

5 Initialization

5.1 Variables and Entities

Initialization affects not just variables but the more general notion of "entity". An entity is any name in the program that represents possible values at run time. This covers:

- *Variables*: local variables of routines, attributes of classes (each representing a field in the corresponding instances).
- "*Read-only*" entities: manifest constants, as in the declaration *Pi*: *REAL* = 3.141592, formal arguments of routines, **Current** representing the current objects (similar to this or self).

A variable *x* can be the target of an assignment, as in *x* := *y*. Read-only entities can't, as they are set once and for all. More precisely: a constant has a fixed value for the duration of the program; **Current** is set by the execution (for the duration of a call *x*.*f*, the new current object will be the object attached to *x*, as evaluated relative to the previous current object); formal arguments are attached to the value of the corresponding actuals at the time of each call, and cannot be changed during the execution of that call.

Local variables include a special case, the predefined local **Result** denoting the result to be returned by a function, as in the following scheme:

```
clicked_window (address: URL) : WINDOW                            /2/
                -- Window showing URL for address: depending on user
                -- request, either same as current display window or
                -- newly created one.
        do
                if must_open_in_new_window then
                        create Result.make (address)
                else        -- Keep current window, but display address
                        Result := display_window.displaying (address)
                end
        end
```

This example also illustrates the creation instruction, here using the creation procedure *make*. Unlike the constructors of C++, Java or C#, creation procedures in Eiffel are normal procedures of the class, which happen to be marked as available for creation (the class lists them in a clause labeled **create**).

The example also shows a typical context in which the initialization issue arises: *WINDOW* being an attached type, we must make sure that **Result** is attached (non-void) on exit. Clearly a creation instruction (first branch) produces an attached result. The second branch will work too if the function *displaying*, returning a *WINDOW* and hence required to produce an attached result, satisfies this requirement.

5.2 Self-initializing Types

In earlier versions of Eiffel, initialization has always been guaranteed for all variables, to avoid the kind of run-time situation, possible in some other languages, where the program suddenly finds a variable with an unpredictable value as left in memory by the execution of a previous program if any. This would be a reliability and security risk. Any solution to the initialization issue must continue to avoid that risk.

Since read-only entities are taken care of, it remains to ensure that every variable has a well-defined value before its *first use*, meaning more precisely:

- For local variables of a routine *r*, including **Result** for a function: the first use in any particular call to *r*.
- For attributes: the first use for any particular object. This doesn't just mean the first use in a routine call *x*. *r* (...) where *r* is a routine of the class: it can also be during a creation operation **create** *x*. *make* (...) at the time the object is being created, where *make* may try to access the attribute; or, if contract monitoring is on, in the evaluation of the class invariant, before or after the execution of a routine call.

Eiffel's earlier initialization rules were simple:

1 A variable of a reference type was initialized to *Void*. This policy will be retained for detachable types, but we need a different one for attached types; this is the crux of our problem.

2 The basic types *BOOLEAN*, *CHARACTER*, *INTEGER*, *REAL*, all of them expanded types, specify default initialization values, respectively **False**, null character, 0, 0.0.

3 Programmer-defined expanded types were required to include *default_create* among their creation procedures. *default_create* is a procedure defined in class *ANY* (the top-level class of which all other classes are descendants, similar to Object in other frameworks but in the context of multiple inheritance) where it does nothing; any class can redefine it to implement a specific initialization scheme. Although implicitly present in every class, *default_create* is not necessarily available as a creation procedure; this happens only if the class lists it explicitly in its **create** clause.

Case 2 is in fact an application of case 3, assuming proper versions of *default_create* in the basic types. Note that *default_create* only needs to create a new object in the case of reference types; for variables of expanded types, it can simply apply its algorithm to an existing object.

It is tempting to keep this *default_create* requirement for expanded types, extend it to attached types, and declare victory. This was, however, found too restrictive. First, it would break most existing code: as noted above, we would like to assume that most application classes do not need void values, and so can effortlessly be reinterpreted, under the new scheme, as attached; but we cannot assume that all or even a majority already support *default_create* as creation procedure. In fact this is not such a common case since most non-trivial class invariants require creation procedures with arguments. Even for new classes, the *default_create* requirement is not one we can easily impose on all application programmers.

Even if we can't use impose it universally, this requirement does address the initialization problem for variables of the corresponding types, so we may rely on it when applicable. We give such types a name:

Definition: Self-initializing type

A type is **self-initializing** if it is one of:

- A detachable type.
- A type (including the basic types) based on a class that makes *default_create* from *ANY* available for creation.

For variables of self-initializing types we adopt a policy of **lazy initialization**. The previous policy was systematically to initialize object fields (corresponding to attributes) on object creation, prior to the execution of any creation procedure such as *make* above, and local variables on routine entry, using in both cases the default value, language-set or provided by *default_create*. Instead, we can now afford a more flexible policy: no sweeping general initialization, but, on first access to a variable of a self-initializing type, check whether it has already been set; if not, call *default_create*. This actually implies a slight change of semantics for expanded types:

- Under the previous rules, the semantics for expanded types was that a variable directly denoted an object of that type, rather than a reference. For an attribute, this means a **subobject** of the current object; for a local variable, the compiler-generated code may allocate the object directly on the **stack** rather than on the heap. One of the disadvantages of this approach, apart from its too greedy approach to initialization with *default_create*, is that it requires a special rule prohibiting cycles in the client relation between expanded types: if both *A* and *B* are expanded classes, you can't have *A* declare an attribute of type *B* and conversely, since this would mean that every object of type *A* has a subobject of type *B* and conversely.
- The new semantics is simply that expanded types simply represent objects with **copy semantics** rather than the default *reference semantics*. Using such an object as source of an assignment will imply copying, rather than assign a reference.
- As a result, the clumsy prohibition of no client cycles between expanded classes goes away.
- We also removed the requirement that expanded types provide *default_create* for creation; in other words, they do not have to be self-initializing. When they are not, the same alternative initialization techniques as for attached reference types, discussed below, are available to them, and the same lazy initialization semantics.
- Compilers can now implement expanded types through references; this is purely a matter of implementation, as the only requirement is copy semantics.
- In the vast majority of cases, there are indeed no cycles in the client relation; compilers can then optimize the representation by using subobjects and stack-based allocation as before. In the general spirit of the language's evolution, the idea is to make things simpler and more easy to learn for programmers (just talk about copy semantics, don't worry about implementation), remove hard-to-justify restrictions, and expect a little more of the compiler writer.

- Previously, a creation instruction **create** *x.make* (...), where *make* can be *default_create*, would not (as noted) create an object for expanded *x*, but simply apply *make* to an existing stack object or subobject. Now it may have to create an object, in particular if the relation does have cycles. This is an implementation matter not affecting the semantics.
- Whether or not it actually creates an object, the creation instruction will be triggered the first time the execution needs a particular expanded variable. This change from a greedy policy (initialize everything on object creation or routine entry) to a lazy one can break some existing code if *make* or *default_create* performs some significant operations on the current object and others: this initialization can occur later, or not at all. The new policy seems better, but maintainers of existing software must be warned of the change and given a backward-compatibility option to keep the old semantics.

Except for copy semantics, the rest of this discussion applies to self-initializing reference types as well as to expanded types.

To summarize the results so far, we have narrowed down the initialization problem by taking care of one important case: self-initializing types, for which the policy will be to create the object (or possibly reinitialize an existing object in the expand case) if its first attempted use finds it uninitialized.

This leaves — apart from generic parameters — the case of non-self-initializing types.

5.3 Self-initializing Attributes

If the type is not self-initializing, we can make an individual *attribute* (instance variable) self-initializing. (The technique will not be available for local variables.)

Here, especially for readers steeped in C++ or its successors such as Java and C#, a little digression is necessary about what I believe to be a misunderstanding of object-oriented principles in a specific aspect of the design of these languages. They consider an attribute (also called *instance variable*, *member variable* or *field*) as fundamentally different from a function (or *method*); this is illustrated by the difference in call syntax, as in

```
y := x.my_attribute                                          /3/
```

versus

```
y := x.your_function ()          -- Note the parentheses     /4/
```

which makes it impossible to change your mind — go from a storage-based implementation to a computation-based one for a certain query returning information on objects of a certain type — without affecting *every single client* using the query in the above styles. The Principle of Uniform Access **[8]** requires instead that such a choice of implementation should not be relevant to clients. In Eiffel (as already in Simula 67) the syntax in both cases is simply

```
x.her_query
```

which could call either an attribute or a function; the term "query" covers both cases.

The problem goes further. Because a class in C++ etc., when it exports an attribute, exports the information that it is an attribute (rather than just a query), it exports it for both reading and writing, permitting remote assignments to object fields, such as

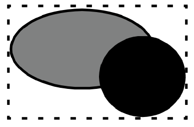
```
        x.my_attribute = new_value                                    /5/
```
This scheme is widely considered bad practice since it violates the principles of information hiding and data abstraction, which would require a procedure call

```
        x.set_my_attribute (new_value)                                /6/
```
with a proper set_my_attribute procedure. As a result, textbooks warn against exporting attributes — always a bad sign, since if a language design permits a construct officially considered bad the better solution would be to remove it from the language itself — and suggest writing instead an exported function that will return the value of the attribute, itself declared secret (private), so that instead of the plain attribute access **/3/** one will call, in style **/4/**, a function whose sole purpose is to access and return the secret attribute's value. But this leads to lots of noise in the program text, with secret attributes shadowed by little functions all of the same trivial form (one line to return the value). "Properties", as introduced by Delphi and also present in C#, handle such cases by letting the programmer associate with such a secret attribute a "getter" function and a "setter" procedure, which will respectively return the value and set it. The advantage is to permit the assignment syntax /5/ with the semantics of a procedure call /6/ (as also now possible in Eiffel, with examples below). But the price is even more noise: in C#, altogether three keywords (value, set, get) in the language, and still two separate features in the class — the attribute and the property — for a single query.

The Eiffel policy is different. The Uniform Access Principle suggests that we should make as little difference as possible between attributes and functions. Each is just a query; if exported, it is exported as a query, for access only. The interface of a class (as produced by automatic documentation tools) doesn't show the difference between an attribute and a function; nor, as we have seen above, does the call syntax (no useless empty parentheses).

Standard Eiffel goes further in the application of the principle. In particular, it was previously not possible, largely for fear of performance overhead, to redefine an attribute into a function in a descendant class (while the reverse was permitted). Partly as a result, attributes could not have contracts — preconditions and postconditions — as functions do; postcondition properties can be taken care of in the class invariant, but there is no substitute for preconditions. These restrictions are now all gone, in part because of the availability of better implementation techniques that avoid penalizing programs that don't need the extended facilities. With a new keyword **attribute**, one can equip an attribute with a contract:

```
bounding_rectangle: RECTANGLE                                         /7/
                -- Smallest rectangle including whole of current figure
        require
                bounded
        attribute
        ensure
                Result.height = height
                Result.width = width
                Result.lower_left = lower_left
                Result.contains (Current)
        end
```



With this convention the attribute can freely be redefined into a function and conversely. Note that **Result**, previously meaningful for functions only, is now available for attributes too; the example uses it for its postcondition. This further enhances the symmetry between the two concepts. The previous syntax for declaring an attribute, *x*: *SOME_TYPE*, remains available as an abbreviation for

```
x: SOME_TYPE
        attribute
```

End of digression. This new generality of the concept of attribute suggests another simple mechanism taking care of explicit attribute initialization, and making attributes even more similar to functions: give them an optional algorithm by allowing instructions after **attribute**, the same way a function has instructions after **do** (see e.g. /2/). So we can for example provide *shadow* with an explicit initialization:

```
bounding_rectangle: FIGURE                                              /8/
                -- Smallest rectangle including whole of current figure
                -- (Computed only if needed)
        require
                bounded
        attribute
                create Result.set (lower_left, width, height)
        ensure
                -- As above
        end
```

The semantics is to call this code if — and only if — execution finds, for a particular object, the attribute uninitialized on first use of that object.

An interesting benefit of this technique is to provide a "**once per object**" mechanism, letting us performing a certain operation at most one time on any object, and only when needed, in a lazy style. That's what the algorithm for *bounding_rectangle* does. Here is another example, from a class *COMPANY_STOCK*:

```
stock_history: LIST [VALUATION]                                         /9/
                -- Previous valuations over remembered period
        attribute
                if {l: LIST [VALUATION]}
                                database.retrieved (ticker_symbol) then
                        Result := l   -- Yields list retrieved from database
                else
                        create Result       -- Produces empty list
                end
        ensure
                -- ...
        end
```

The stock history list might be huge, so we only want to retrieve it into memory from the database for a particular company if, and when, we need it. This could be done manually by keeping a boolean attribute that says whether the list has been retrieved, but the technique is tedious is there are many such "lazy" queries. Self-initializing attributes solve the problem in a simpler way. Note the use of an Object Test to check whether the object structure retrieved from the database is of the expected type.

The presence of self-initialization for a particular attribute will, in the semantics, take precedence over self-initialization at the class level if also present.

This concept of self-initializing attribute further narrows down the initialization issue. But it does not yet solve it completely:

- It does not apply to local variables. In fact we could devise a similar notion of "self-initializing local", where the declaration includes an initialization algorithm. But this seems overkill for such a narrowly-scoped notion.
- For both attributes and local variables the requirement of self-initialization cannot be the only possibility. In some cases a human reader sees immediately that for every use of a variable at run time an assignment or creation will have happened before, giving it a well-defined attached value. Then the lazy initialization-on-demand of either self-initializing types or self-initializing attributes is not necessary, and would in fact be deceptive in the program text since the initialization code will be never be executed. We should simply let things go as originally written, after checking that there is no risk of undefined or void value.

5.4 Certified Attachment Patterns

The last observation leads to the third and last initialization technique: rely on compilers (or other static checking tools) to verify that explicit assignment or creation will have occurred before every use. The authors of Spec# have reached a similar conclusion, taking advantage of modern compiler technology; they write **[1]**:

Spec# stipulates the inference of non-nullity for local variables. This inference is performed as a dataflow analysis by the Spec# compiler.

We differ from this assessment in only one respect: it is not possible in Eiffel to refer to "the compiler". There are a number of Eiffel compilers, and one of the principal purposes of the ECMA standard is precisely to keep maintaining their specific personalities while guaranteeing full syntactical, validity and semantic interoperability for the benefit of users. Even if there were only one compiler as currently with Spec#, we do not wish to let programmers depend on the smartness of the particular implementation to find out — by trying a compilation and waiting for possible rejection — if a particular scheme will work or not. There should be precise rules stating what is permissible and what is not. These rules should be available in a descriptive style, like the rest of a good language specification, not in an operational style dependent on the functioning of a compiler. They should be valid for any implementation; after all, much of the progress in modern programming language description has followed from the decision to abstract from the properties of a particular compiler and provide high-level semantic specifications instead.

Apart from this difference of view, the Eiffel rules result from the same decision of relying — for cases not covered by self-initializing types or attributes — on statically enforceable rules of good conduct. We call them Certified Attachment Patterns:

Definition: Certified Attachment Pattern (CAP)

A Certified Attachment Pattern for a non-self-initializing variable *x* is a general program context in which *x* is guaranteed to be non-void.

Here is a typical Certified Attachment Pattern, for an arbitrary attribute or local variable *x*. If the body of the routine starts with a creation instruction or assignment of target *x*, then the immediately following instruction position is a CAP for *x*. This is a very important pattern; in fact (as the reader may have noted) neither of the last two examples /8/ /9/ would be valid without it, because they rely on a **create Result** ... instruction to ensure that **Result** is non-void on return from the attribute evaluation. This property is trivial — since the **create** instruction is the last in the routine — but without the CAP there would be no way to rely on it.

The stock history example /9/ also relies on another CAP: if *cap1* and *cap2* are two Certified Assignment Patterns for *x*, then so is **if** *c* **then** *cap1* **else** *cap2* **end** for any condition *c*.

Here is a third CAP, assuming that *x* is a local variable or formal routine argument:

```
if x /= Void then                                                    /10/
        ... Any Instructions here, except for assignments of target x.
end
```

The **then** branch is a CAP for *x*. It would **not** be a valid CAP if *x* were an attribute, as the "Instructions" could include procedure calls that perform an assignment (of a possible void value) to *x*. But for a local variable we can ascertain just by looking locally at the **then** branch that there is no such assignment.

Certified Attachment Patterns, from the above definition, apply to "non-self-initializing variables". This includes variables of attached types that are not self-initializing, but also variables of **detachable** types, which we had not considered for a while. In fact, as the reader may have noted, /10/ is meaningful only for a detachable type; if the type of *x* is attached, and not self-initializing, then the attempt to evaluate it in the test *x* /= *Void* of /10/ would not work; and the test is meaningless anyway for *x* of an attached type. But for detachable *x* the CAP is useful, as it allows us to perform a call of target *x* as part of the Instructions.

Such calls are indeed valid. The Target Validity Rule, the basic constraint of the void-safe type system, stated that "A qualified call *a*.*f* is only valid if *a* is attached". As noted, this usually means that the type of *a* is attached, but we can generalize the definition to take advantage of CAPs:

Definition: Attached expression

An expression *a* is attached if and only if either:

- Its type is an attached type.
- It occurs as part of a Certified Attachment Pattern for *a*.

Without this CAP, we would have, for every use of a local variable *x* of a detachable types, to write an Object Test (with the need to shadow *x* with an explicitly declared Object-Test-Local *y*, as in **if** {*y*: *TYPE_OF_X*} *x* **then** ...) every time we want to use *x* as target of a call. Occasion ally this cannot be avoided, but often the routine's algorithm naturally includes **if** *x* /= *Void* **then** ..., which the CAP allows us to use as it stands, in the way we would normally do.

An associated CAP is for *x* in the else part of **if** *x* = *Void* **then** ... **else** ... end. Another one for *x*, particularly important for class invariants, is in *other_condition* in

```
x /= Void and then other_condition
```

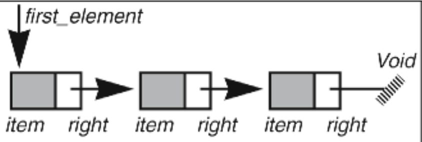
where **and then** is the nonstrict conjunction operator, guaranteeing that the second operand will not be evaluated if the first evaluates to false. This also works if we replace **and then** by **implies** (implication, nonstrict in Eiffel, i.e. *a* **implies** *b* is defined with value true if *a* has value false, even if *b* is not defined); it works for **or else** if we change the test to *x* = *Void*.

Another Certified Attachment Pattern, similar to the first, is particularly important for loops iterating on linked data structure. It is of the form

```
from
        ...
until
        x = Void
loop
        ... Any Instructions not assigning to x ...
end
```

If *x* is a local variable (again, not an attribute), it remains attached throughout the Instructions. This makes possible, without further ado — in particular, without any Object Test — a whole range of common traversal algorithms, such as this one for searching in a linked list:

```
from
        x := first_element
until
        x = Void or else Result
loop
        Result := (x.item = sought_value)
        x := x.right
end
```



(**Result** starts out false; the loop will set it to true if and only if the item in one of the list cells has an *item* field equal to *sought_value*. *x* is as before a local variable) The CAP enables us to write the loop exactly as we would write it anyway, with the guarantee that it will not produce any void call. A look the previous version of the EiffelBase library suggests that many existing loops will similarly compile and run "as is"; occasionally, application of the Target Validity rule will require a slight rewrite, at worst inclusion of some instructions in an Object Test. This is extremely encouraging (especially given the complexity of some of the intermediate suggestions, some involving changes to the loop construct, that were experimented before we arrived at the general solution reported here). More generally, we see as particularly attractive the prospect of replacing, in such a library, hundreds of occurrences of

```
some_routine (x: SOME_TYPE)
        require
                x /= Void
                x.some_property
```

by just

```
some_routine (x: SOME_TYPE)
        require
                x.some_property
```

with the non-void test turned into a compile-time guarantee (*SOME_TYPE* being an attached type) that *x* indeed represents an object, so that we can concentrate on the more meaningful contractual properties such as *x.some_property*.

A CAP, very useful in practice, applies to the instructions that immediately follow a series of creation instructions **create** *a* ..., for one or more *a*: these instructions are a CAP for such *a*. Beyond local variables, this also applies to attributes, somewhat neglected by the previous CAPs, and enables us to handle many simple cases such as guaranteeing that a just created **Result** of a function, as in /8/ and /9/, is attached as expected.

Finally, as a concession to programmers who prefer to run the risk of an exception in the case of a variable that shouldn't be void but is, we include as CAP the position immediately following

```
check
        x /= Void
end
```

taking advantage of Eiffel's **check** instruction. This instruction will raise an exception if *x* is void. This CAP is an escape valve, as we do not feel like preventing programmers from using an exception-based style if that's their choice (which we may disapprove).

Using CAPs to guarantee attachment is a pessimistic policy, erring, if at all, on the side of safety: if we cannot absolutely guarantee the impossibility of a void value, the Target Validity rule will (except, as noted, under backward-compatibility compiler options, to avoid breaking existing code) reject the code. The design rule for CAPs is not that they *support all correct cases*, but that they *reject any incorrect case*. We can afford to miss some correct cases if they do not occur too frequently; the only drawback will be that programmers may have, in some extreme and (we hope) rare situations, an Object Test that appears unnecessary. (Remember that one of the reasons those cases are so rare is that CAPs are only a technique of next-to-last resort, and Object Tests of the last one: in many practical cases the Eiffel programmer can rely on self-initializing types or variables.) As a result we can afford not to care too much about cases that worry the Spec# designers **[3] [1]**, such as a creation procedure that needs to access an attribute that one is not sure has already been initialized. In Eiffel, the attribute will often be of a self-initializing type, or itself be declared **attribute** ... so that it is self-initializing; if not, there might be a matching CAP; if not, the programmer can always get away with an Object Test or, if that's the preferred style, force a CAP with a **check** instruction as above. We don't have to turn our compilers into prodigies of dataflow analysis.

We do not, in fact, want CAPs to be too sophisticated. They should cover situations where it is immediately obvious to a human reader (and, besides, true!) that an expression cannot take on a void value even though it is neither of an attached type nor self-initializing. The argument should be simple and understandable. If it is convoluted, it may be just as well to force a slight rewrite of the immediate context to make the safety argument compelling. In other works, when it comes to establishing guaranteed attachment status, *we do not want Eiffel compilers to be too smart* about possible voidness. The argument should always remain clearly understandable to the reader of the program, in the Eiffel spirit of clarity and quality-focused software engineering. (There is still a great need for sophisticated dataflow analysis and more generally for very smart compiler writers: generate the fastest and most compact code possible.)

This approach rests under the assumption that a small number of simple CAPs capture the vast majority of practical situations. This seems to be the case with the set of CAPs sketched above, covering most of what has been included in the Eiffel standard, where they are of course specified much more precisely. On the organizational side, the existence of an international standards committee provides a good framework: even if the CAP catalog remains separate from the Eiffel standard proper, permitting more frequent additions, it should remain subject to strict quality control and approval by a group of experts after careful evaluation. Technically (beyond "proof by committee"), the goal should be, with the development of a proper mathematical framework, to *prove* — through machine-validated proofs — the validity of proposed CAPs. The three criteria that must remain in force throughout that process are:

- A guarantee of correctness beyond any doubt.
- Simple enforceability by any reasonable compiler, *without* dataflow analysis.
- Understandability of all CAPs by any reasonably qualified programmer.

6 Object Tests and Their Scopes

The Object Test form of boolean expression, {*x*: *T*} *exp*, was presented in the Overview, which gave the essentials. *T* is an attached type; *exp* is an expression; *x* is a fresh name not used for any entity in the enclosing context, and is known as the **Object-Test-Local** of the expression. Evaluation of the expression:

- Yields true if and only if the value of *exp* is attached to an object of type *T* (and so, as a particular consequence, not void).
- Has the extra effect of binding *x* to that value for the subsequent execution of the program extract making up the scope of the Object Test. *x* is a Read-Only entity and hence its value can never be changed over that scope.

The scope depends on where the Object Test appears. We saw that in **if** *ot* **then** ... **else** ... **end**, with *ot* an Object Test, the scope is the **then** part. Also, if a condition is of the form *ot* **and then** *boolexp* or *ot* **implies** *boolexp*, the scope includes *boolexp* as well. With a negated Object Test, **not** {*x*: *T*} *exp*, the scope, in a conditional instruction, is the **else** part; such negated variants are particularly important for loops, since in

```
from ... until not {x: T} exp loop ... end
```

the whole loop clause — the loop body — is part of the scope.

The notion of scope has been criticized by some experienced Eiffel programmers who in line with the Eiffel method's emphasis on command-query separation **[8]** do not like the idea of an expression evaluation causing initialization of an entity as a side effect. But apart from some unease with the style there seems to be nothing fundamentally wrong there, and the construct does provide a useful and general scheme.

In particular, it is easier to use than Eiffel's earlier Assignment Attempt mechanism *x* ?= *y*. Although an effective and widely used method of run-time type ascertainment, the Assignment Attempt treats the non-matching case by reintroducing a void value (for *x*), which in light of this entire discussion doesn't seem the smartest idea. An Assignment Attempt almost always requires declaring the target *x* specially as a local variable; with Object Test we integrate the declaration in the construct. It should almost always be followed by a test *x* /= *Void*, yet it is possible for programmers to omit that test if they think the object will always match; this is a source of potential unreliability. Here we essentially force such a test through the notion of scope.

In general, the Object Test seems an attractive alternative to the various run-time type identification and ascertainment (including downcasting) in various languages; it seems to subsume them all.

7 Generic Classes

Perhaps the most delicate part of the attachment problem is the connection with genericity. There turns out to be a remarkably simple solution. (This needs to be pointed out from the start, because the detailed analysis leading to that solution is somewhat longish. But the end result is a four-line rule that can be taught in a couple of minutes.)

Consider a container class such as *ARRAY* [*G*] (a Kernel Library class) or *LIST* [*G*]. *G* is the "**formal generic parameter**", representing an arbitrary type. To turn the class into a type, we need to provide an "**actual generic parameter**", itself a type, as in *ARRAY* [*INTEGER*], *LIST* [*EMPLOYEE*]. This process is called a "**generic derivation**". The actual generic parameter may itself be generically derived, as in *ARRAY* [*LIST* [*EMPLOYEE*]].

Genericity can be constrained, as in *HASH_TABLE* [*ELEMENT*, *KEY* –> *HASHABLE*] which will accept a generic derivation *HASH_TABLE* [*T*, *STRING*] only if *STRING* conforms to (inherits from) the library class *HASHABLE* (in the Eiffel Kernel Library it does). Unconstrained genericity, as in *ARRAY* [*G*], is formally an abbreviation for *ARRAY* [*G* –> *ANY*].

None of these class declarations places any requirement on the attachment status of a type. You can use — subject to restrictions discussed now — *ARRAY* [*T*] as well as *ARRAY* [**?** *T*]. The same holds even for constrained genericity: attachment status does not affect conformance of types. (So if *U* inherits from *T*, **?** *U* still conforms to *T*. It's only for entities and expressions that the rules are stricter: With *x*: *T* and *y*: **?** *U*, *y* does not conform to *x*, prohibiting the assignment *x* := *y*.) Without such rules, we would have to provide two versions of *ARRAY* and any other container class: once for attached types, one for detachable types. Not an attractive prospect.

Now consider a variable of type *G* in a generic class *C* [*G*]. What about its initialization **?** *G* stands for an arbitrary type: detachable or attached; if attached, self-initializing or not. Within the class we don't know. But a client class using a particular generic derivation needs to know! Perhaps the most vivid example is array access. Consider the declarations and instruction

```
x, y: T
i, j: INTEGER
arr: ARRAY [T]
...
arr.put (x, i)          -- Sets entry of index i to x; Can also be     /11/
                        -- written more conventionally as arr [i] := x
```

This sets a certain entry to a certain value. Now the client may want to access an array entry, the same or another:

```
y := arr.item (j)       -- Can also be written as y := arr [j]         /12/
```

T is an attached type. Instruction /11/ will indeed store an attached value into the *i*-th entry, assuming the array implementation does its job properly. Since the class *ARRAY* [*G*] will, as one may expect, give for function *item* the signature

```
item (i: INTEGER): G
```

and the actual generic parameter for *arr* is *T*, instruction /12/ correspondingly expects the call *arr*.*item* (*j*) to return a *T* result, for assignment to *y*. This should be the case for *j* = *i*, but what about other values of *j*, for which the entry hasn't been explicitly set by a *put* yet?

We expect default initialization for such items of container data structures, as for any other entities. But how is class *ARRAY* [*G*], or any other container class, to perform this initialization in a way that will work for all possible actual generic parameters:

detachable, as in *ARRAY* [**?** *T*], expanded, or attached as with *ARRAY* [*T*] but with *T* either self-initializing or not?

The tempting solution is to provide several versions of the class for these different cases, but, as already noted, we'd like to avoid that if at all possible. We must find a way to support actual generic parameters that are detachable, easy enough since we can always initialize a *G* variable to Void, or attached, the harder case since then we must be faithful to our clients and always return an attached result for queries such as *item* that yield a *G*.

The result of such a query will be set by normal instructions of the language, for example creations or assignments. For example the final instruction of a query such as *item* may be **Result** := *x* for some *x*. Then **Result** will be attached if an only if *x* is attached. Although *x* could be a general expression, the properties of expressions are deducible from those of their constituents, so in the end the problem reduces to guaranteeing that a certain entity *x* of the class, of type *G*, is attached whenever the corresponding actual parameter *T* is. Let's consider the possible kinds of occurrence of *x*:

G1 x may be a formal argument of a routine of C. From the conformance rules, which state that only *G* itself conforms to *G*, *x* will be of type *T* (the actual generic parameter of our example), detachable or attached exactly as we want it to be. Perfect! Other cases of read-only entities are just as straightforward. From then on we consider only variables.

G2 We may be using *x* as a target of a creation instruction **create** *x*.*make* (...) or just **create** *x*. That's the easiest case: by construction, *x* will always be attached, regardless of the status of *T*. (To make such creation instructions possible the formal generic parameter must satisfy some rules, part of the general Eiffel constraints: it must specify the creation procedures in the generic constraint, as in *C* [*G* –> *C* **create** *make* **end**], where *make* is a procedure of *C*, or similarly *C* [*G* –> *ANY* **create** *default_create* **end**]. Then the actual generic parameter *T* must provide the specified procedures available as creation procedures.)

G3 We may be using *x* as the target of an assignment *x* := *y*. Then the problem is just pushed recursively to an assessment of the attachment status of *y*.

G4 The last two cases generalize to that of an occurrence in a Certified Assignment Pattern resulting from the presence of such a creation instruction or assignment instruction guaranteed to yield an attached target, for example at the beginning of a routine.

G5 So the only case that remains in doubt is the use of *x* — for example in the source of an assignment — without any clear guarantee that it has been initialized. If *x*'s type were not a formal generic, we would then require *x* to be self-initializing: either by itself, through an **attribute** clause, or by being of a self-initializing type. But here — except if we get a self-initializing attribute *x* of type *G*, a possible but rare case — we expect the guarantee that *G* represents a self-initializing type.

We don't have that guarantee in the general case; *T*, as noted, may be of any kind. And yet if *T* is not self-initializing we won't be able to give the client what it expects. So what we need, to make the mechanism complete, is language support for specifying that a generic parameter must be self-initializing (that is to say, as defined earlier, either detachable or providing *default_create* as a creation procedure). The syntax to specify this is simply to declare the class, instead of just *C* [*G*], as

```
class C [? G] ...
```

This syntax is subject to criticism as it reuses a convention, the **?** of detachable types, with a slightly different meaning. But it seemed preferable to the invention of a new keyword; it might change if too many people find it repulsive, but what matters here is the semantic aspect, captured by the validity rule:

Generic Initialization rule

Consider a formal generic parameter *G* of a class C.

1. If any instruction or expression of C uses an entity of type *G* in a state in which it has not been provably initialized, the class declaration must specify **?** *G* rather than just *G*.
2. If the class declaration specifies **?** *G*, then any actual generic parameter for *G* must be self-initializing.

A formal generic parameter of the form is known as a **self-initializing formal**; clearly we must add this case to the list of possibilities in the definition of self-initializing types.

In the Standard these are two separate validity rules. There are both very easy to state and apply. The first is for the authors of generic classes — typically a relatively small group of programmers, mostly those who build libraries — and the second for authors of clients of such classes; they're a much larger crowd, typically including all application programmers, since it's hard to think of an application that doesn't rely on generic classes for arrays, lists and the like.

Class *ARRAY* will fall under clause 1, declared as *ARRAY* [**?** *G*]; this makes it possible to have arrays of *T* elements for an attached type *T*. The rule is very easy to explain to ordinary application programmers (the second group): *ARRAY* gives you a guarantee of initialization — you'll never get back a void entry from an *ARRAY* [*T*], through *arr*. *item* (*i*), or arr [*i*] which means the same thing —, so you must provide that default initialization yourself by equipping *T* with a *default_create*. Now if you can't, for example if *T* is really someone else's type, then don't worry, that's OK too: instead of an *ARRAY* [*T*] use an *ARRAY* [**?** *T*]; simply don't expect arr [*i*] to give you back a *T*, it will give you a **?** *T*, possibly void, which you'll have to run through an Object Test if you want to use it as attached, for example as the target of a call. Fair enough, don't you agree?

This **?** *G* declaration leading to a requirement of self-initializing actual generic parameters applies to class *ARRAY* because of the specific nature of arrays, where initialization has to sweep through all entries at once. It doesn't have to be carried through to data structures subject to finer programmer control. For example, in class *LIST* [*G*] and all its EiffelBase descendants representing various implementations of sequential lists, such as *LINKED_LIST* [*G*], *TWO_WAY_LIST* [*G*], *ARRAYED_LIST* [*G*] etc., the basic operation for inserting an item is *your_list.extend* (*x*), adding *x* at the end, with implementations such as

```
extend (x: G)
            -- Add x at end.
      local
            new_cell: LINKABLE [G]
      do
            create new_cell.make (x)
      end
```

Then, to get the items of a list, we access fields of list cells, of type *LINKABLE* [*G*] for the same *G*, through queries that return a *G*. This is case G1, the easiest one, in the above list, guaranteeing everything we need to serve our attached and detachable clients alike.

Most generic classes will be like this and will require no modification whatsoever, taking just a *G* rather than a **?** *G*. *ARRAY* and variants (two-dimensional arrays etc.) are an exception, very important in practice, and of course there will be a few other cases.

8 Getting Rid of Catcalls

Having completed Void Call Eradication, we come to the second major problem, whose discussion (to reassure the reader) will be significantly shorter; not that the problem is easier or less important, but simply because the solution will almost trivially follow from the buildup so far.

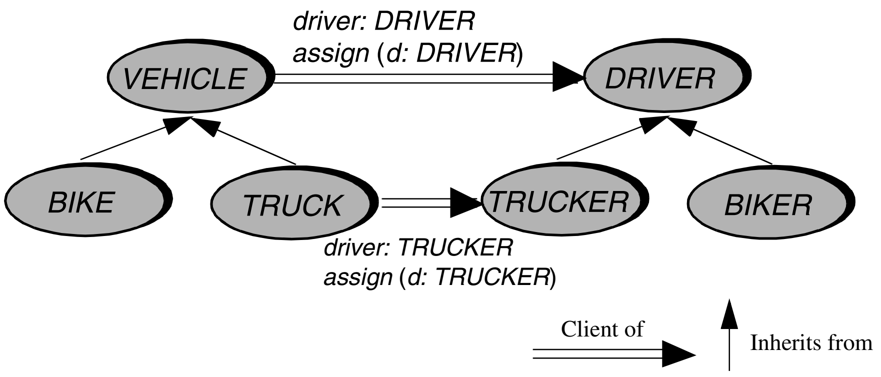
Typed object-oriented programming languages are almost all *novariant*: if you redefine a routine in a descendant of the class containing its original declaration, you cannot change its signature — the type of its arguments and results.

And yet... modeling the systems of the world seems to require such variance. As a typical example, consider (see the figure on the opposing page) a class *VEHICLE* with a query and command

```
driver: DRIVER
register (d: DRIVER) do driver := d end                              /13/
```

A vehicle has a driver, of type *DRIVER* (a companion class) and a procedure *register* that assigns a driver. No we introduce descendant classes *TRUCK* and *BICYCLE* of *VEHICLE*, and *TRUCKER* and *BIKER* of *DRIVER*. Shouldn't *driver* change type, correspondingly, in *TRUCK* and *BICYCLE*, to *TRUCKER* and *BIKER* respectively? All signs are that it should. But novariance prevents this.

The policy that would allow such type redefinitions is called covariance (from terminology introduced by Luca Cardelli); "co" because the redefinition follows the direction of inheritance.



In fact there is no type risk associated with redefining **query results**, such as *driver*, covariantly. Still, most languages don't permit this, probably because then programmers wouldn't understand why they can also redefine **routine arguments** covariantly. If you redefine *driver*, you will also want to redefine register so that its signature reads

```
register (d: TRUCKER) do driver := d end          -- in TRUCK   /14/
register (d: BIKER)   do driver := d end          -- in BIKE    /15/
```

and so on. Eiffel allows you to do this and in fact provides an important abbreviation; if you know ahead of time (that is to say, in the ancestor class) that an entity will be covariant, you can avoid redefinitions altogether by declaring the entity from the start as "anchored" to another through the **like** keyword: here in *TRUCK* you can replace /13/ by

```
register (d: like driver) ... Rest as before ...    -- in VEHICLE /16/
```

where the **like** type declaration anchors *d* to driver, so that the redefinitions of /14/ and /15/ are no longer needed explicitly (but the effect is the same). **like**, avoiding explicit "redefinition avalanche", is the covariant mechanism par excellence.

With covariant arguments we have a problem **[9]** because of polymorphism and dynamic binding. The declarations and call

```
v: VEHICLE
d: DRIVER
...
v.register (d)
```

look reasonable enough; but what the call is preceded by the assignments

```
v := some_truck
d := some_biker
```

with the types of the assignment sources as implied by their names? We end up assigning to a truck a driver qualified only to ride a bike. Then when the execution attempts, on an object of type *TRUCKER*, to access a feature of the driver — legitimately assumed to be a truck driver, on the basis of the redefinition —, for example *driver*.*license_expiration_date*, we get a crash, known as a catcall (assuming truck licenses expire, but bike licenses don't). This is the reason novariance is the general rule: even though catcalls happen rarely in well-written programs, they are just as much of a risk as void calls.

The solution proposed here is simple: force the programmer who makes a covariant argument redeclaration to recognize the risk through the following rule:

Covariant argument redeclaration rule

The type of a covariant argument redeclaration, or of an anchored (like) argument declaration, must be detachable.

In our example an explicit redeclaration will have to be written, instead of /14/

```
register (d: ? TRUCKER) ...                -- in TRUCK        /17/
```

and an anchored one, instead of /16/:

```
register (d: ? like driver) ...            -- in VEHICLE      /18/
```

This requires the body of the routine (in the redefined version for the first case, already in the original version for the second case) to perform an explicit Object Test if it wants to apply a call to the argument, ascertaining it to be of the covariantly redefined type. Catcalls clearly go away.

The semantics of **?** U in the covariant redefinition of an argument *x* originally of type *T* is slightly different from the usual one involving possible void values. It really means "from *T* down to *U*". It also requires a particular convention rule for the semantics of a new precondition clause of the form require else *x*.*some_U_property* (we interpret it as {*y*: *U*} *x* **and then** *y*.*some_U_property*). So there is a certain amount of kludginess on the theoretical side. But in practice the technique seems to allow us to keep covariance for expressiveness, while removing the dangers.

This technique is not so far from what programmers instinctively do in languages such as C++, Java and C# which enforce novariance. The modeled system, as noted, often cries for covariance. So in the descendant class the programmer will introduce a variable of the new type, the one really desired, and "downcast" (the equivalent of an Object Test) the novariant argument to it. One finds numerous examples of this pattern in practical code from the languages cited. The above rule leads us to a similar solution, but it is more explicit and, one may argue, better for modeling realism: the programmer specifies, in the redefinition, the "true" new type of the argument (like *TRUCKER*); the type system accepts his covariant behavior, but forces him to recognize the risk to of that behavior to others around him, specifically to "polymorphic perverts" (callers of the original routine which, through polymorphism, actually use the new argument type disguised under the old one), and to handle that risk by checking explicitly for the type of the actual objects received through the formal argument.

9 An Application to Concurrency

The third major problem to which the ideas discussed here provide a solution is lazy object reservation in concurrent object-oriented programming.

Concurrency is badly in need of techniques that will make concurrent programs (multithreaded, multi-processed, networked, web-serviced...) as clear and trustworthy as those we write for sequential applications. Common concurrent mechanisms, most notably thread libraries, still rely on 1960-era concepts, such as semaphores and locks. Deadlocks and data races are a constant concern and a not so infrequent practical occurrence.

An effort to bring concurrent programming to the same level of abstraction and quality that object technology has brought to the sequential world led to the definition of the SCOOP model of computation (Simple Concurrent Object-Oriented Programming **[11]**, with a first implementation available from ETH). This is not the place to go through the details of SCOOP, but one aspect is directly relevant. If an entity *x* denotes a "**separate**" object — one handled by a thread of control, or "processor", other than the processor handling calls to the current object — it appears essential to permit a call of the form *x*.*f* only if *x* is an argument of the enclosing routine *r*. Then a call to *r*, with the corresponding actual argument *a* representing a separate object, will proceed only when it has obtained exclusive access to that object, and then will retain that access for the duration of the call. Coupled with the use of preconditions as wait conditions, this is the principal synchronization mechanism, and leads to elegant algorithms with very little explicit synchronization code (see for example the Dining Philosophers in **[11]**).

The rule then is:

Validity and semantics of separate calls

A call on a separate object is permitted only if the object is known through a formal argument to the enclosing routine.

Passing the corresponding separate actual arguments to the routine will cause a wait until they are all available, and will reserve them for the duration of the call.

A specific consequence of this policy is of direct interest for this discussion:

Object Reservation rule

Passing a separate actual argument a to a routine *r* reserves the associated object.

The corresponding formal argument *x* in *r* must also be declared as **separate**, so that it is immediately clear, from the interface of *r*, that it will perform an object reservation.

So far this has implied the converse rule: if a is separate, *r* (*a*) will wait until the object is available, and then will reserve it.

But this second part is too restrictive. If *r* doesn't actually include any call *x*. *f* where *x* is the formal argument corresponding to *a*, we don't need to wait, and the policy could actually cause deadlock. For example in

```
keep_it: separate T                -- An attribute

r (x: separate T)
            -- Remember a for later use.
      do
            keep_it := x
      end
```

we just use *x* as source of an assignment. The actual calls will be done later using *keep_it*, which we will have to pass (according to the above validity rule) as actual argument to some other routine, which performs such calls. But in a call to this *r* we don't need to wait on *x*, and don't want to.

This could be treated as a *compiler optimization*: the body of *r* doesn't perform any call on *x*, so we can skip the object reservation. But this is not an acceptable solution, for two reasons:

- The semantics — including waiting or not — should be immediately clear to client programmers. They will see only the interface (signature and contracts), not the **do** clause.
- In a descendant class, we can redefine *r* so that it *now* performs a call of target *x*. Yet under dynamic binding a client could unwittingly be calling the redefined version!

We need a way to specify, as part of the official routine interface, that a formal argument such as *x* above is, although separate, non-blocking.

The solution proposed is: **use a detachable type**, here **? separate** *T*. With the declaration

```
r (x: ? separate T)                -- The rest as above
```

no reservation will occur.

With this policy, whether reservation occurs is part of the routine's specification as documented to client programmers. The rule is consistent with the general property of detachable and attached types: we need *x* to be attached only if we are going to perform a call on it.

In the example, we can only retain the assignment to *keep_it* if this attribute is itself detachable. If it is attached, declared as **separate** *T* rather than **? separate** *T*, we must rewrite the assignment as

```
if {y: separate T} x then
        keep_it := y
end
```

with the obvious semantic rule that **an object test of separate type causes reservation of the object**. Unlike in the case of an argument, this rule is acceptable since no information hiding is involved: we are looking at implementation, not a routine's interface.

The technique also fits well with inheritance. If *r* had an argument of type **? separate** *T*, we cannot of course redefine it as **separate** *T* in a descendant (the reverse is, as always, possible). But if the descendant version does need to perform a call on *x*, whereas the original didn't, it can achieve the result through an Object Test:

```
if {y: separate T} x then
        y.some_operation
end
```

which will cause a wait — not on entry to the routine (which could contradict the semantics advertised to the client) but as part of that routine's implementation.

It appears then that the distinction between attached and detachable types, and the general-purpose Object Test with its semantics adapted to the concurrent (separate) case, solve this particular problem of concurrent object-oriented programming too.

10 Conclusion

It was impossible to resist including the self-citation that opens this article, but hard to resist the temptation of removing the parts that don't quite fit, especially the bit about the two or three minutes. The ideas presented here didn't come with the self-evidence of morning mist; it was more like the icy rain of an endless Baltic winter. Yet the mechanism is indeed a minuscule syntax extension, the **?** symbol (even if used with two slightly different semantics), combined with the replacement of an existing instruction, the assignment attempt, by a simpler and more general mechanism, the Object Test. With a few validity rules that any reasonable program should meet without the programmer thinking much about them — even though the presentation in this article may have appeared long-winded since it took into account many details, special cases, compatibility issues and the rationale for every decision — the result does address several major problems in one sweep; one of these problems, the starting point for the whole effort, is the only remaining source of crashes in typed object-oriented programs, and hence of critical practical importance. The second one has also been the subject of a considerable literature. The third one is less well known, but of importance for concurrent applications. And in passing we have seen that for two issues that had been addressed by previous mechanisms — Run-Time Type Identification, possible in many languages, and "once per object", for which Eiffel already had a solution — the mechanisms allows new techniques that may offer at least an incremental improvement on those already known.

So while it is for the reader to judge whether the citation is arrogant, I do hope that the mechanisms presented above, as available in Standard Eiffel, will have a lasting effect on the quality of software that we can produce, using the best of object technology.

References

[1] Mike Barnett, Rustan Leino and Wolfram Schulte: *The Spec# Programming System*; CASSIS proceedings, 2004.

[2] Craig Chambers et al., papers on the Self language at http://research.sun.com/self/papers/papers.html.

[3] Manuel Fähndrich and Rustan Leino: *Declaring and Checking Non-null Types in an Object-Oriented Language*; in OOPSLA 2003, SIGPLAN Notices, vol. 38 no. 11, November 2003, ACM, pp. 302-312.

[4] ECMA Technical Committee 39 (Programming and Scripting Languages) Technical Group 4 (Eiffel): *Eiffel Analysis, Design and Programming Language*, Draft international standard, April 2005.

[5] Erik Meijer and Wolfram Schulte: *Unifying Tables, Objects, and Documents*; in Proc. DP-COOL 2003, also at http://research.microsoft.com/~emeijer/Papers/XS.pdf.

[6] Bertrand Meyer: *Eiffel: The Language*, Prentice Hall 1990 (revised printing 1991). See also **[12]**.

[7] Bertrand Meyer: *Reusable Software: The Base Object-Oriented Component Libraries*, Prentice Hall, 1994.

[8] Bertrand Meyer: *Object-Oriented Software Construction, 2nd edition*, Prentice Hall, 1997.

[9] Bertrand Meyer, reference **[8]**, chapter 17: "*Typing*".

[10] Bertrand Meyer, *Prelude to a Theory of Void*; in *Journal of Object-Oriented Programming*, vol. 11, no. 7, November-December 1998, pages 36-48.

[11] Bertrand Meyer, reference **[8]**, chapter 30: *Concurrency, distribution, client-server and the Internet.*

[12] Bertrand Meyer, *Standard Eiffel*, new edition of **[6]**, in progress.

The Emergent Structure of Development Tasks

Gail C. Murphy[1], Mik Kersten[1], Martin P. Robillard[2], and Davor Čubranić[3]

[1] Department of Computer Science, University of British Columbia
murphy@cs.ubc.ca, beatmik@acm.org
[2] School of Computer Science, McGill University
martin@cs.mcgill.ca
[3] Department of Computer Science, University of Victoria
cubranic@cs.uvic.ca

Abstract. Integrated development environments have been designed and engineered to display structural information about the source code of large systems. When a development task lines up with the structure of the system, the tools in these environments do a great job of supporting developers in their work. Unfortunately, many development tasks do not have this characteristic. Instead, they involve changes that are scattered across the source code and various other kinds of artifacts, including bug reports and documentation. Today's development environments provide little support for working with scattered pieces of a system, and as a result, are not adequately supporting the ways in which developers work on the system. Fortunately, many development tasks do have a structure. This structure emerges from a developer's actions when changing the system. In this paper, we describe how the structure of many tasks crosscuts system artifacts, and how by capturing that structure, we can make it as easy for developers to work on changes scattered across the system's structure as it is to work on changes that line up with the system's structure.

1 Introduction

The tools that developers use to build a large software system provide an abundance of information about the structure of the system. Integrated development environments (IDEs), for example, include views that describe inheritance hierarchies, that present the results of system-wide searches about callers of methods, and that report misuses of interfaces. These IDEs have made it easier for developers to cope with the complex information structures that comprise a large software system.

However, some of our recent work suggests that the focus on providing extensive structural information may be having two negative effects on development:

- developers may be spending more time looking for relevant information amongst the morass presented than working with it [10], and
- developers may not always be finding relevant information, resulting in incomplete solutions that lead to faults [4, 19].

We believe that these problems can be addressed by considering how a developer works on the system. More often than not, development tasks require changes that are scattered across system artifacts. For instance, a developer working on a change task might change parts of several classes, may read and edit comments on parts of a bug report, may update parts of a web document, and so on. As a developer navigates to and edits these pieces, a structure of the task emerges.

In this paper, we describe how this structure crosscuts the structures of system artifacts and we explore how the capture and description of task structure can be used to present information and support operations in an IDE in a way that better matches how a developer works. We believe support for task structure can improve the effectiveness of existing tools and can enable support for new operations that can improve a developer's individual and group work.

We begin with a characterization of how tasks crosscut system artifacts, providing data about the prevalence of scattered changes and arguing that the changes have structure that is crosscutting (Sect. 2). We then introduce a working definition of task structure (Sect. 3) and describe what an IDE with task structure might provide to a developer (Sect. 4). We then elaborate on the possibilities, explaining some of our initial efforts in making task structure explicit (Sect. 5), discuss some open questions (Sect. 6), and describe how our ideas relate to earlier efforts (Sect. 7).

2 Tasks Crosscut Artifacts

Building a software system involves many different kinds of tasks. A one-year diary study of 13 developers who were involved in building a large telecommunications system found 13 different kinds of tasks, including estimation, high-level design, code, and customer documentation [15]. In this paper, we focus on *change tasks* that affect the functionality of the system in some way, by fixing bugs, improving performance, or implementing new features. To simplify the discourse in this paper, we use the term *task* to mean change task.

To complete a task, a developer typically has to interact with several kinds of artifacts, including source code, bug descriptions,[1] test cases, and various flavours of documentation. Conceptually, these artifacts form an information space from which an IDE draws information to display to a developer. Since the source code tends to form the majority of the structure, we focus our characterization mainly on it, returning to the more general information space in later sections of the paper.

2.1 Occurrence of Scattered Changes

It has long been a goal of programming language and software engineering research to make it possible to express a system such that most modification tasks require only localized changes to a codebase [14]. To achieve this goal, modularity mechanisms have been introduced into the programming languages we use (e.g., classes in object-oriented languages) and various design practices have evolved (e.g., design patterns [6]). Despite these advances, we contend that the completion of many tasks still requires changes that are scattered across a code base.

[1] Bug descriptions, or reports, at least in many open-source projects, are used to track not just faults with the system, but enhancements and other desired changes.

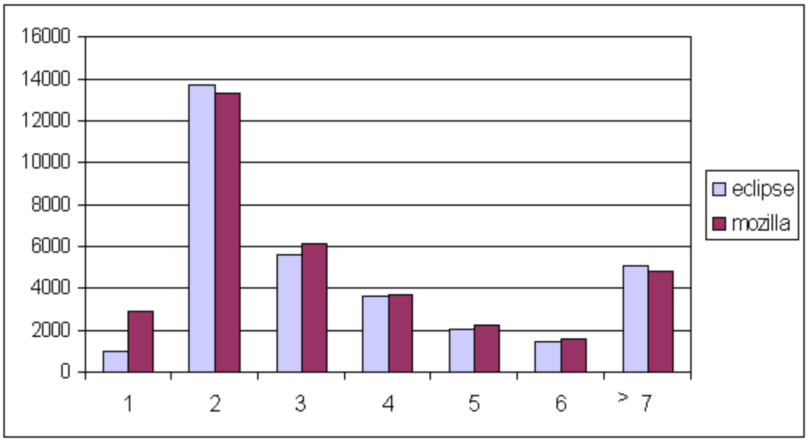


Fig. 1. The number of files (x-axis) involved in check-in transactions (y-axis) for Eclipse and Mozilla

To illustrate that many changes have this property, Fig. 1 shows the number of files checked-in as part of transactions from two large open-source projects — Eclipse and Mozilla.[2] Following a common heuristic used for open-source projects, a transaction is defined as consisting of file revisions that were checked in by the same author with the same check-in comment close in time [13]. For both of these systems, over 90% of the transactions involve changes to more than one file.

To provide some insight into the relationship between the changes and the structure of the system, we randomly sampled 20 transactions that involved four files from the Eclipse data. Of these transactions, fifteen involved changes in multiple classes located close together (i.e., within the same package). These changes are scattered, but it might be considered that they are contained within some notion of module (i.e., a Java package). However, five transactions included changes across packages, and of these five, two included changes across more than one plug-in (a significant grouping of related functionality in Eclipse). Assuming that a transaction roughly corresponds to a task,[3] a reasonable number of tasks (25% of those sampled) involved changes scattered across non-local parts of the system structure.

2.2 Crosscutting Structure of Changes

Are scattered changes simply the result of a bad system structure or is there some structure to the scattering? To provide some insight into these questions, we consider a typi-

[2] The Eclipse project can be found at `eclipse.org` and the Mozilla project can be found at `mozilla.org`. The check-in data for Eclipse comes from 2001/04/28 until 2002/10/01 and the check-in data for Mozilla comes from 1998/03/27 until 2002/05/08. Only data for transactions involving 20 or less files is shown.

[3] This is a reasonable assumption because of the work practices used in developing this open-source system.

cal change in the Eclipse code base. We chose to use an example from Eclipse because it is generally considered to be well-designed and extensible. We follow Eclipse documentation guidelines in the approach we take to implementing the change.

The task of interest involves a change to a hypothetical Eclipse plug-in to support the editing and viewing of an HTML document. This HTML plug-in provides an outline view that displays the structure of an HTML document as a tree, where the headings and paragraphs are nodes in the tree. Imagine that your task is to modify the outline view of the HTML plug-in to add nodes that represent hyperlinks.

To perform this task, you need to update both the HTML document model and the view. Assuming the recommended structure for Eclipse plug-ins, this means changing methods in a `ContentProvider`, a `LabelDecorator` and a `Selection-Listener` class. You also need to add a menu action and update appropriate toolbars which requires modifying another class. In addition, you need to declare the new action and any associated icon in an XML file (i.e., `plugin.xml`). In total, this simple change task involves modifications scattered across four Java classes, two parts of an XML file, and an icon resource.

Although these changes are scattered, there is *structure* to the change task; the structure happens to *crosscut* multiple parts of multiple artifacts. In simple terms, two structures crosscut each other if neither can fit neatly inside the structure provided by the other [12]. A developer well-versed in Eclipse plug-in development would be able to explain this structure, and much of it is recorded in the documentation about how to extend Eclipse. The structure of the source code has been chosen to make adding a new listener to a view a change that is localized in the structure, whereas adding a brand-new element (as in our task) is a change that crosscuts the structure.

We believe that many of the tasks involving scattered changes are not ad hoc, but that they do have a crosscutting structure. In our work, we have found that this crosscutting structure emerges from how a developer works with the code base [19, 10]. In the remainder of the paper, we show how this structure, once made explicit, can be used to make IDEs work better for developers.

3 Task Structure

To ground our discussion, we introduce a simple working definition of task structure.

> A **task structure** consists of the parts of a software system and relationships between those parts that were changed to complete the task.

Conceptually, consider forming a graph based on information found in all of the artifacts comprising the system. In this graph, the nodes are structural parts of the artifacts and the edges are relationships between those parts. The structure of a task consists of a collection of subgraphs from this graph. Each node in the graph includes information about the artifact in which it appears, the name of the part, and the type of the part: each kind of artifact will have its own types of parts. For example, the types of parts found in a Java class include method definitions, field definitions, and inner class definitions. As another example, the types of parts found in a bug report include dates when the report is opened or closed, and text fields with discussions about the report. Each edge in the

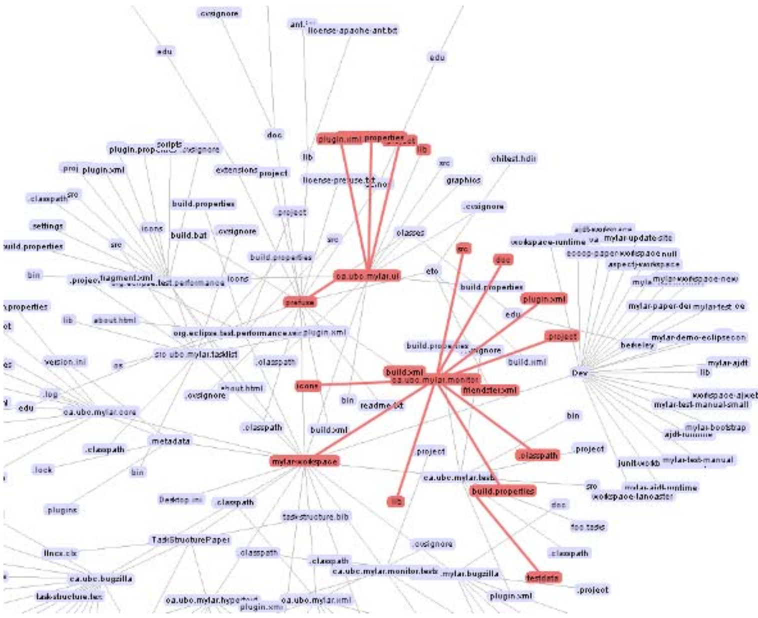


Fig. 2. A graph showing parts of artifacts and relationships between the parts comprising a simple system. The highlighted portions represent the structure of one task performed on the system

graph includes information about the artifact in which it appears (if any) and the type of the relationship. Some relationships will be defined explicitly in a project's artifacts, whereas others may be inferred by tools. For example, a call between two methods in a Java program appears explicitly in the source code, whereas a relationship that indicates a file revision helped solve a particular bug may be inferred by a tool [3].

As a concrete example, we return to the task of adding a new element to an existing view in an Eclipse plug-in. Figure 2 shows a portion of the graph of parts and relationships from artifacts comprising the system.[4] Even though only a fraction of the structure of a small number of artifacts is included in the graph, the amount of information is overwhelming. However, only small parts of the graph relate to the task; the highlighted nodes and edges in Fig. 2 form the structure of the task.

Our definition of task structure is based on completed tasks. An advantage of this definition is that the task structure can be determined with certainty if the time points at which the task started and finished are known. However, we also want to make use of task structure as a task is being performed. We use *task context* as a means of approximating task structure and as a means of describing the subgraphs of the information space of interest when performing a task.

A **task context** consists of parts and relationships of artifacts relevant to a developer as they work on the task.

[4] The graph was generated using prefuse [9].

This definition of task context relies on the concept of *relevance* of parts of a system to a task. Relevance can be defined in a number of ways, all of which include some element of cognitive work on the part of a developer [26]. A simple way to determine relevance is for a developer to manually mark the parts and relationships as relevant as they are exploring code [20]. Automatic determinations of relevance are also possible. For example, we are investigating two approaches in which relevance is based on the interaction of the developer with the information in the environment, such as which program elements are selected. In one approach, the interaction information is used to build a model of the degree to which a developer is interested in different parts of the system [10]. This degree-of-interest model is then used to predict interest in other elements and in related project artifacts. In the second approach, relevance is determined by analyzing the interaction information according to the frequency of visits to a program element, the order of visits, the navigation mechanism used to find an element (e.g., browsing, cross-reference search, etc.), and an analysis of the structural dependencies between elements visited [21].

In the rest of this paper, we assume that task structure and task context information is available and focus on providing some examples of how it might be used to improve a developer's work environment.

4 Improving a Developer's Work with Task Structure

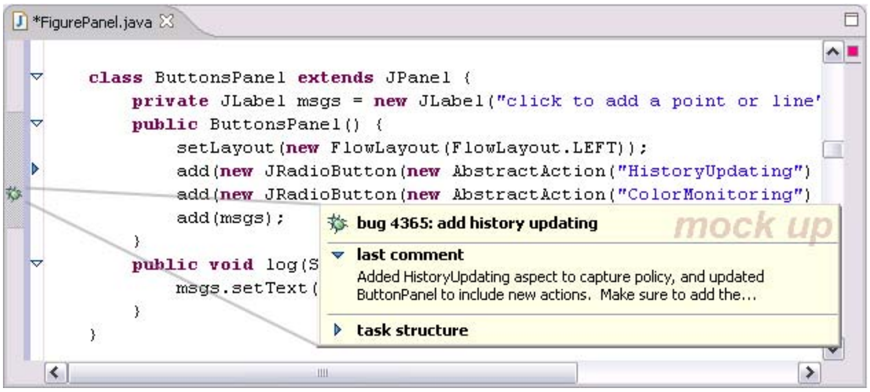
Imagine that you are a developer working with an IDE that includes support for capturing, saving and operating on task contexts and task structures. In this section, we describe what it might be like to use this IDE to work on a development task. As we indicate in the scenario, several features we describe have been built or proposed as part of earlier efforts. Task structure enables these operations to be more focused and to provide more semantic information, without any significant input from the developer.

The system on which you are working allows a user to draw points and lines in a window, and to change their colour.[5] The system also has a mode, which when set through a radio button, supports the undo of actions taken by the developer through the user interface.

Your current task involves adding support to enforce the use of a predetermined colour scheme in a drawing. A radio button is to be provided to turn the colour scheme enforcement on and off. When the enforcement is on, the colours of points and lines in the drawing are to be modified to meet the colour scheme and any subsequent request to change a colour will be mapped to the colour scheme.

You start the task by navigating through some of the code attempting to find relevant parts. As you navigate, the IDE is building up your task context based on your selections and edits. After some navigation you determine that you need to add some code in the `ButtonsPanel` class to add in the necessary radio button. At this point, your task context includes information about several methods you have visited and the constructor in `ButtonsPanel`. As you add in the call to add a new radio button, a green bug icon

[5] This example is based on a simple figure editor used to teach the AspectJ language [11].



```
class ButtonsPanel extends JPanel {
    private JLabel msgs = new JLabel("click to add a point or line'
    public ButtonsPanel() {
        setLayout(new FlowLayout(FlowLayout.LEFT));
        add(new JRadioButton(new AbstractAction("HistoryUpdating")
        add(new JRadioButton(new AbstractAction("ColorMonitoring")
        add(msgs);
    }
    public void log(S
        msgs.setText(
    }
}
```

bug 4365: add history updating

last comment

Added HistoryUpdating aspect to capture policy, and updated ButtonPanel to include new actions. Make sure to add the...

task structure

Fig. 3. Based on the task context, a green bug icon appears indicating another bug report may be relevant to the task being performed

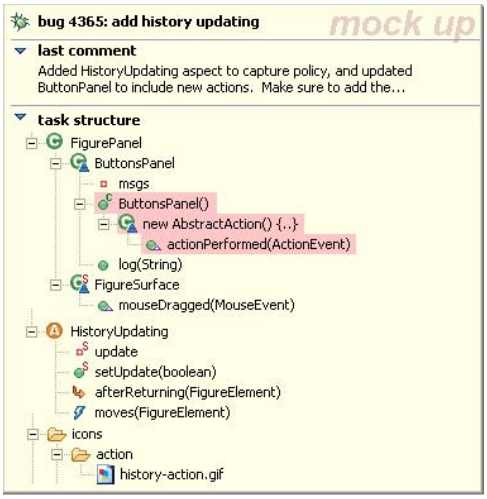
appears in the left gutter of the editor (Fig. 3).[6] This icon appears because a tool in the IDE, running in the background, has determined that there is a completed change task whose task structure is similar to your task context.

You decide to click on the bug icon. A popup window appears that describes some information about the bug (Fig 3). You read the description of the bug and you realize that it is similar to the task on which you are working.[7] Since the related bug has been resolved, it has an associated task structure. You expand this task structure and the tree view of the structure shows you which parts overlap with your task context (the highlighted nodes in Fig. 4). You notice the `HistoryUpdating` aspect listed in the task structure that supports the undo functionality. You have not considered whether you will use an aspect to complete your task. However, you look at the code for the aspect and realize that it implements similar functionality to what is needed for your task. After considering the options, you decide to use an aspect-oriented approach and you create a `ColourControl` aspect based on the `HistoryUpdating` aspect. Guided by the previous task context, you also add an image for the new action to the system.

Before you check-in the code for your completed task, you want to ensure your changes will not conflict with concurrent changes being made by other members of your team. As you check-in your code using the facilities of the IDE, you select an option to compare your task structure with any task contexts that your team members have made available (by selecting an option in the IDE). The IDE tool supporting this comparison looks for overlap between your task structure and your team members task contexts and if it finds overlap, it considers any effect, using static analyses, each task has had on the overlapping parts.

[6] The user interfaces described are mock-ups of how the described functionality might be provided.

[7] This type of functionality is similar to our Hipikat tool [3]. We sketch the differences between the Hipikat approach and using task structure for this purpose in Sect. 5.



bug 4365: add history updating

mock up

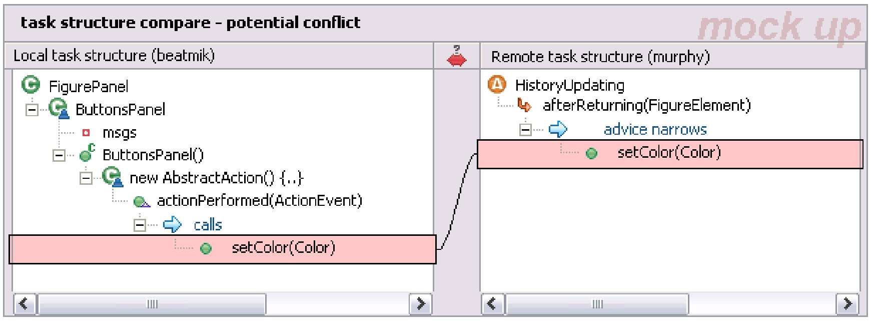
last comment

Added HistoryUpdating aspect to capture policy, and updated ButtonPanel to include new actions. Make sure to add the...

task structure

```
FigurePanel
  ButtonsPanel
    msgs
    ButtonsPanel()
      new AbstractAction() {..}
        actionPerformed(ActionEvent)
    log(String)
  FigureSurface
    mouseDragged(MouseEvent)
HistoryUpdating
  update
  setUpdate(boolean)
  afterReturning(FigureElement)
  moves(FigureElement)
icons
  action
    history-action.gif
```

Fig. 4. The task structure of the completed task with highlights indicating overlap with the current task context



task structure compare - potential conflict

mock up

Local task structure (beatmik)

```
FigurePanel
  ButtonsPanel
    msgs
    ButtonsPanel()
      new AbstractAction() {..}
        actionPerformed(ActionEvent)
          calls
            setColor(Color)
```

Remote task structure (murphy)

```
HistoryUpdating
  afterReturning(FigureElement)
    advice narrows
      setColor(Color)
```

Fig. 5. A comparison of your task structure with a team member's task context identifies a possible conflict. The conflict is determined by statically analyzing the effect of each task context and comparing the results

Figure 5 shows the results of the comparison for the task you are about to complete. It shows that part of your local task structure includes a call to a `setColor` method (left side of Fig. 5). It also shows that one of your team member's task context's has modified the `HistoryUpdating` aspect to add advice that narrows `setColor` [17]. Narrowing advice may result in the `setColor` method not being called under some circumstances. Given this information, you can contact your colleague to determine how to resolve the conflicts between your changes.[8]

[8] This type of fine-grained conflict determination has similarities to soft locking in Coven [2].

As noted, several of the features that task structure makes possible in this hypothetical scenario have been proposed previously. In comparison to these existing approaches, task structure provides three benefits over existing approaches:

1. it can be determined with minimal effort from the developer as it emerges from how the developer works on the system,
2. it provides a conceptual framework and model that can be built into an IDE to make task-related tools easier to build, and
3. it provides information that may be used to focus views in the IDE, allowing a greater density of relevant information to be displayed.

5 Making Use of Task Structure

The scenario described in the last section illustrates how explicit support of task structure in an IDE can benefit a developer. In this section, we elaborate on these points and describe more possibilities. Through this section, we use the term task structure to simplify the discourse as it should be clear when the use of task context would be more precise.

5.1 Improving IDE Tools

An ideal IDE would present the information a developer needs, when it is needed, and with a minimum of interaction from the developer. Such an IDE would reduce the amount of time a developer spends trying to find relevant information. We outline four ways that an IDE with support for task structure could help move towards this goal.

Reducing Overload in Views and Visualizations. IDEs present system structure mostly in lists and tree views, with some graphical visualizations [24]. When used on large systems, these existing presentation mechanisms tend to overload the developer with information, making it difficult to find the information of interest. For example the Package Explorer, a commonly used tree view in Eclipse which shows the decomposition of Java source into packages, files, classes, and other structural elements, often contains tens of thousands of nodes when used on a moderately-sized system (e.g., see the left-hand side of Fig. 6; notice that the tree structure is not visible). A task's structure can be used to determine what information should be made more conspicuous to a developer. For instance, the task structure can be highlighted [22]. Or, the task structure can be used to filter the view so as to show only task-relevant information as is the case in our Mylar prototype (see the right-hand side of Figure 6; the bolded parts are the elements that are the most important to the task) [10]. Either way, the views can make it clear to the developer the elements important to the task.

Scoping Queries. Task structure can be used to scope the execution of code queries performed by a developer. A default setting, for instance, may be to query code only within one or two relationships in the overall graph of structural information (Sect. 3) from the code on which the query is invoked. Scoping queries with task structure could have two potential benefits: queries may execute more quickly for large systems, and the information returned may be more relevant, reducing the time needed for a developer to wade through search results.

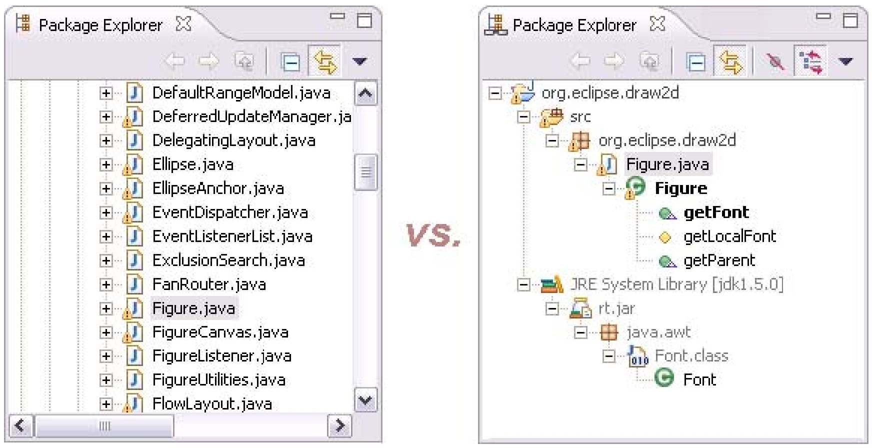


Fig. 6. A view of the containment hierarchy of a system without Mylar active (left-hand side) and with Mylar active (right-hand side). In the Mylar view, the focus provided by task context enables the relevant information to fit on the screen without a scrollbar, and enables the structural relationships to be visible

Performing Queries Automatically. In addition to scoping queries, parts in the task structure can be used to seed queries that run automatically. We call these active queries and they could be used to seed active views [10]. For example, the active search view in the Mylar prototype eagerly finds and displays all Java, XML, and bug reports related to parts in the task structure [10]. This kind of view provides a developer with the structural information they need when it is needed. The result could be a reduction in the number of interruptions a developer must typically make to think about and formulate a query, as the most relevant queries are formulated and executed automatically. Active queries also do away with the need to wait on query execution, since queries are executed automatically in the background.

The concept may also be helpful in the implementation of the IDE. The Eclipse IDE, for instance, requires the Abstract Syntax Tree (AST) for a class to be in memory in order to support features such as semantic highlighting.[9] However, operations that span multiple files, such as the rename method refactoring, are time consuming and require the developer to wait until all related ASTs are loaded into memory. The task structure could be used to define the slices of ASTs that should be kept in memory in order to make common refactorings instantaneous (i.e., as quick for changes across files as they are for changes within the file).

Supporting Task Management. IDEs provide little to no support for managing the tasks a developer is performing. The best support may be an ability to read and manage bug reports within the IDE. Task structure can improve this situation. For example, as part of our Mylar project, we have prototyped support for enabling developers to

[9] Semantic highlighting refers to the ability to highlight code according to properties such as whether the code is an abstract method invocation, a reference to a local variable, etc.

associate task structures with specific tasks and to switch between them. Mylar can then filter the views of the system according to the selected task. The task structure can also be attached to a bug report, enabling a developer to re-start if they return to the bug at a later time. The task structure, in effect, is a form of externalization of the developer's memory of the task.

5.2 Improving Collaboration

Over the lifetime of a system many developers work on many tasks. We believe that communicating this structure to other developers as they work, and storing it as the system evolves, can provide collaborative tools with an effective representation of group memory.

Forming and Accessing a Group Memory. It is not uncommon when working on a software development project to come across a problem that is reminiscent of a past problem with the system that has since been solved (Sect. 4). In earlier work, we demonstrated the benefits of processing the artifacts comprising a software system to form a group memory that may then be searched for relevant information as a developer is performing a task [3, 4]. One benefit is that developers may be more aware of subtle, but relevant, information. For example, in an experiment we conducted, newcomers to a project took into account additional information presented from the group memory and finished an assigned task more completely than experts who did not have access to the group memory [4]. Our previous work treated task structure implicitly, forming links automatically between parts of related artifacts. Explicitly stored task structures enable more focused comparisons between a past system and a current system and allow new operations across the group memory, such as an analysis tool that could identify all of the third party APIs involved in commonly reported defects.

Sharing Task Structure. Task structure encapsulates a developer's knowledge about the system. As discussed above, developers may want to store this knowledge in order to access it at a later time. In addition, they may want to share it with others. For example, a developer delegating a task could include the task structure in order to help the team member pick up the task where it was left off. Sharing of task structure could also be done in real-time in order to make developers aware of the activities of their team members. For example, in an open-source project where team members are distributed across time zones, knowing the parts of the system that have been worked on by others can encourage dialog and prevent merge problems. In comparison to existing approaches to providing such awareness [23], task structure can enable a deeper comparison, seeding automated handling by tools or discussions between involved developers with more information.

5.3 Improving the IDE Platform

In addition to improving the developer's experience, task structure may help solve issues related to a number of tools provided by an IDE, and may help simplify the development of tools.

Capturing and Recommending Workflow. In this paper, we have focused on information overload that developers face when working on the content of large systems. These developers also face information overload in the user interfaces of IDEs. Enterprise-application development tools, such as IBM's Rational Software Architect, offer sophisticated support for development across the lifecycle, which results in dozens of views and editors, and hundreds of user interface actions. It can be difficult for developers to know what features exist, let alone try to find them. Adaptive interfaces [7] and Eclipse's capabilities[10] address this in a general way, based on aggregate information about how features are used. We see potential for making the user interface more aware of the task being performed by capturing the task structure of developers who use these tools effectively, and then mining this information for task-specific interaction patterns of the user interface. Mined patterns may suggest ways to focus the user interface on only those tools needed for the completion of a particular task [22].

Simplifying Tool Development. IDE platforms such as Eclipse make it easy to build new tools that expose system structure. For example, a new view that shows all methods overriding the currently-selected method is easy to add. In our experience, it is harder to add tools that depend on some notion of task, and each tool must develop its own ad hoc model of task. While it is possible to layer task information on the models provided by the IDE through an index over existing elements and relationships, we see potential for task information to be more central. For example, it would be beneficial to be able to tag an element as being part of some named task, and to then be able to trigger an action based on when an attempt is made to synchronize that element with the repository. Task structure information could also be used to arbitrate user interface issues; for example, a tool might use a task's structure to determine which of several competing annotations are most applicable to show in the gutter of an editor.

6 Open Questions

Our working definition of task structure is simple and extensional. These characteristics make it easy to describe the possibilities of task structure and do not unduly constrain what a task is or how developers work on tasks. It is an open question as to whether this definition is too simple. It may be that tools built on this definition require information about why artifacts were changed, or the order in which they were changed, to provide meaningful information to a developer. It may also be necessary to include in the definition notions of what constitutes a task, whether a task is worked on in one time period or across various blocks of time, amongst others. These questions will need both empirical and formal investigation.

Regardless of the programming language and software engineering technologies used, we believe many change tasks have an emergent crosscutting structure because it is impossible to simultaneously modularize a system for all kinds of changes that may occur. This statement deserves investigation, such as a characterization of task structure

[10] A capability in Eclipse is a feature set that can be enabled or disabled by a user. Capabilities are pre-defined and configured when the IDE is shipped.

for changes performed on systems intended for a variety of domains, written in a variety of languages, of different ages, and so on. It is also an open question as to whether, at this point, more benefit to the developer might result from better support for explicit task structure than new means of expressing sophisticated modularity.

7 Related Work

7.1 Tasks and Desktop Applications

Explicit capture and manipulation of task information has been studied in the domain of desktop applications (e.g., document processors and email clients). Of these, the project most similar to some of our efforts is TaskTracer [5], which is intended to help knowledge workers deal effectively with interruptions, and which seeks to help knowledge workers reuse information about tasks completed in the past. TaskTracer monitors a worker's interaction with desktop application resources, such as mail messages and web documents, attempting to build up a grouping of resources related to a particular task. The worker has to name the task being worked upon when they start the task. Although some of our goals overlap, we differ fundamentally from TaskTracer in our intention to maintain fine-grained structural information across artifacts; TaskTracer works only at the level of resources or files. We believe the collection of fine-grained information provides several benefits. For instance, we can support detailed comparisons about how a current and a past task compare. As a second example, we can trigger the recall of potentially useful information for a task based on the current task context.

7.2 Tasks and Development Environments

In the context of development environments, the term task has largely been used from a tool builder's point of view. For example, the Gandalf project recognized the variety of tasks that needed to be supported by the software development process and created a suite of tools to support the generation of an environment particular to a project [8]. The researchers recognized the need to deal with such issues as expertise of the developer, but focused on the problems that were more important at that time, such as handling the syntax and semantics of the languages being used to develop a system.

More recently, IDEs have introduced user-defined scopes as a way of approximating a concept of task similar to the way that we use the term in this paper. For example, in Eclipse, a developer can define a *working set* which is a set of resources related through the system's containment hierarchy over which queries may be executed and saved. Working sets are more coarse-grained than task structure and a developer must evolve working sets as they change tasks, as opposed to our concept of task structure which evolves from a developer's work.

7.3 Manipulating Program Fragments

The size and complexity of software systems has led to many approaches for extracting and operating on fragments of a system. We describe four approaches that are similar to our idea of task structure as a collection of system fragments. Tarr and colleagues introduced the idea of multi-dimensional software decomposition to support fragments that correspond to concerns [25]. Task structure is similar in cutting across artifacts, but a task need not have the same conceptual coherence that one expects from a concern. In multi-dimensional software decomposition the main operation supported on a fragment is the integration of code into a system, whereas we have considered how task structure supports development-oriented operations, such as work conflicts between team members. Our earlier work on concern graphs is also related to modelling and manipulating fragments of a system's structure that relate to concerns [20]. A main operation supported on a concern graph is the detection of inconsistencies between a version of a system in which a concern graph is defined and an evolved version of the system [18]. The support of inconsistency detection is possible because a concern graph captures more intentional information than task structure. As a third example, virtual source files were proposed to allow a programmer to define an organization for parts of the system appropriate for a task [1]. A virtual source file is defined intentionally based on queries and can be used for such operations as determining conflicting changes between team members (similar to our description in Sect. 4). In contrast to virtual source files, task structure emerges from how a developer works on the system as opposed to requiring the developer to state their intention on a structure of parts of artifacts relevant to a task. Finally, Quitslund's MView source code editor supports the juxtaposition of code elements selected by a query in a single view [16]. Although a fragment in his system is restricted to the results from a set of queries over source code, it shares a similarity with task structure in enabling a development-oriented operation over the fragments, namely enabling editing of the code in a single window. Dealing with fragments in a task-oriented manner may enable better integration of the various fragment ideas into the work environments of developers.

8 Summary

Tools are supposed to make us work more effectively. IDEs have served this purpose for developers in recent years. However, as systems grow more complex, the effectiveness of these development environments is breaking down because they do not adequately support tasks that involve changes to multiple artifacts. In this paper, we have described how many of these tasks do have a structure; the structure emerges from the way in which a developer works with the system. This emergent task structure can be identified and used by an IDE to focus existing views and enable new operations. This support matches the way a developer works, allowing them to modify a system without being overwhelmed by its complexity.

Acknowledgement

Gail Murphy would like to thank AITO for the honour of the Dahl-Nygaard Junior Prize, which made this paper possible. The authors would also like to thank Andrew Black for encouraging a paper to be written, Annie Ying for contributing data and comments, and the inspirational work of Rob Walker, Elisa Baniassad, and Al Lai. The paper is much better for the insightful comments provided by John Anvik, Wesley Coelho, Brian de Alwis, Jan Hannemann, Gregor Kiczales, and Eric Wohlstadter. Projects contributing to the ideas presented in this paper were funded by NSERC and IBM.

References

1. M. Chu-Carroll and J. Wright. Supporting distributed collaboration through multidimensional software configuration management. In *SCM*, volume 2649 of *LNCS*, pages 40–53. Springer, 2001.
2. M. C. Chu-Carroll and S. Sprenkle. Coven: Brewing better collaboration through software configuration management. In *SIGSOFT '00/FSE-8: Proc. of the 8th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering*, pages 88–97. ACM Press, 2000.
3. D. Čubranić and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *ICSE '03: Proc. of the 25th Int'l Conf. on Software Engineering*, pages 408–418. IEEE Computer Society, 2003.
4. D. Čubranić, G. C. Murphy, J. Singer, and K. S. Booth. Learning from project history: a case study for software development. In *CSCW '04: Proc. of the 2004 ACM Conf. on Computer Supported Cooperative Work*, pages 82–91. ACM Press, 2004.
5. A. N. Dragunov, T. G. Dietterich, K. Johnsrude, M. McLaughlin, L. Li, and J. L. Herlocker. TaskTracer: A desktop environment to support multi-tasking knowledge workers. In *IUI '05: Proc. of the 10th Int'l Conf. on Intelligent User Interfaces*, pages 75–82. ACM Press, 2005.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
7. S. Greenberg and I. H. Witten. Adaptive personalized interfaces – a question of viability. *Behaviour and Information Technology - BIT*, 4:31–45, 1985.
8. A. N. Habermann and D. Notkin. Gandalf: software development environments. *IEEE Trans. Software Engineering*, 12(12):1117–1127, 1986.
9. J. Heer, S. K. Card, and J. A. Landay. prefuse: a toolkit for interactive information visualization. In *CHI '05: Proc. of the SIGCHI Conf. on Human Factors in Computing Systems*, pages 421–430. ACM Press, 2005.
10. M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for IDEs. In *AOSD '05: Proc. of the 4th Int'l Conf. on Aspect-oriented Software Development*, pages 159–168, 2005.
11. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP '01: Proc. of the 15th European Conf. on Object-Oriented Programming*, pages 327–353. Springer, 2001.
12. H. Masuhara and G. Kiczales. Modular crosscutting in aspect-oriented mechanisms. In *ECOOP '03: Proc. of the 17th European Conf. on Object-Oriented Programming*, pages 2–28. Springer, 2003.
13. A. Mockus, R. T. Fielding, and J. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Trans. Software Engineering Methodology*, 11(3):309–346, 2002.
14. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
15. D. E. Perry, N. Staudenmayer, and L. G. Votta. People, organizations, and process improvement. *IEEE Software*, 11(4):36–45, 1994.
16. P. J. Quitslund. Beyond files: programming with multiple source views. In *Eclipse '03: Proc. of the 2003 OOPSLA Workshop on Eclipse Technology eXchange*, pages 6–9. ACM Press, 2003.
17. M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *SIGSOFT '04/FSE-12: Proc. of the 12th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering*, pages 147–158. ACM Press, 2004.

18. M. P. Robillard. *Representing Concerns in Source Code*. PhD thesis, University of British Columbia, 2003.
19. M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Trans. Software Engineering*, 30(12):889–903, 2004.
20. M. P. Robillard and G. C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *ICSE '02: Proc. of the 24th Int'l Conf. on Software Engineering*, pages 406–416. ACM Press, 2002.
21. M. P. Robillard and G. C. Murphy. Automatically inferring concern code from program investigation activities. In *ASE '03: Proc. of the 18th Int'l Conf. on Automated Software Engineering*, pages 225–234. IEEE Computer Society Press, 2003.
22. M. P. Robillard and G. C. Murphy. Program navigation analysis to support task-aware software development environments. In *Proc. of the ICSE Workshop on Directions in Software Engineering Environments*, pages 83–88. IEE, 2004.
23. A. Sarma, Z. Noroozi, and A. van der Hoek. Palantr: Raising awareness among configuration management workspaces. In *ICSE '03: Proc. of the 25th Int'l Conf. on Software Engineering*, pages 444–454. IEEE Computer Society, 2003.
24. M.-A. D. Storey, D. Čubranić, and D. M. German. On the use of visualization to support awareness of human activities in software development: A survey and a framework. In *SoftVis '05: Proc. of the 2005 ACM Symp. on Software Visualization*, pages 193–202. ACM Press, 2005.
25. P. Tarr, H. Ossher, W. Harrison, and S. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *ICSE '99: Proc. of the 21st Int'l Conf. on Software Engineering*, pages 107–119. IEEE Computer Society Press, 1999.
26. D. Woods, E. Patterson, and E. Roth. Can we ever escape from data overload? A cognitive system diagnosis. *Cognition, Technology & Work*, 4(1):22–36, 2002.

Loosely-Separated "Sister" Namespaces in Java

Yoshiki Sato[⋆] and Shigeru Chiba

Dept. of Mathematical and Computing Sciences,
Tokyo Institute of Technology
{yoshiki, chiba}@csg.is.titech.ac.jp

Abstract. Most modern programming systems such as Java allow us to link independently developed components together dynamically. This makes it possible to develop and deploy software on a per component basis. However, a number of Java developers have reported a problem, ironically called the *version barrier*, imposed by the strict separation of namespaces. The version barrier prohibits one component from passing an instance to another component if each component contains that class type. This paper introduces a novel concept for Java namespaces, called *sister namespaces*, to address this problem. Sister namespaces can relax the version barrier between components. The main purpose of this paper is to provide a mechanism for relaxing the version barrier, while still allowing type-safe instance accesses between components with negligible performance penalties in regular execution.

1 Introduction

Practically all modern programming environments allow developers to utilize some kind of component system (e.g., JavaBeans [9], EJB [22], CORBA [25], .NET/DCOM/ActiveX [24], Eclipse plug-ins [29]). A component system allows programmers to develop a component-based application, which can be developed and then deployed per component. Most of the component systems for Java adopt a single class loader per component, and thereby create a unique namespace for each application component. A namespace is a map from the class names to the class definitions. A set of classes included in the same component *joins* [1] its own namespace and thus naming conflicts between components can be avoided. Moreover, a component can be dynamically and individually updated without restarting the whole execution environment.

One significant drawback of such component systems for Java is the difficulty for components to communicate across class loader boundaries in the Java Virtual Machine (JVM) [13, 20, 8, 14]. In fact, such communication is well known to frequently cause a cast error `ClassCastException` or a link error `LinkageError`. Most of the link errors are just bugs; an error is caused when a class is wrongly

[⋆] Currently, Mitsubishi Research Institute, Inc., Japan.

[1] A class joining a namespace means it is being loaded by the class loader that creates the namespace.

loaded by both parent and child loaders [16]. These bugs can be easily avoided if developers are careful. However, cast errors are extremely difficult to avoid since this problem is caused by the strict separation of namespaces, ironically called the *version barrier*. The version barrier is a mechanism that prevents a version of a class type from being converted to another version of that specific class type. For instance, it restricts an instance of the former type to be assigned to the variable of the latter type.

In Java, a class type is uniquely identified at runtime by the combination of a class loader and a fully qualified class name. If two class definitions with the same class name are loaded by different loaders, two versions of that class type are created and they can co-exists, although they are regarded as distinct types. The version barrier is a mechanism for guaranteeing that different versions of a class are regarded as different types. This guarantee is significant for performance reasons. If different versions of a class were not regarded as different types, the advantages of being a statically typed language would be lost. Moreover, if the same class definition (i.e., class file) is loaded by different class loaders, different versions of that class are created and regarded as distinct types. Therefore, if two components load the same class file individually, one component cannot pass an instance of that class type to the other.

This paper presents our novel concept of namespaces in Java, which we call *sister namespaces*, and the design of that mechanism. Sister namespaces can relax the version barrier between application components. An instance can be carried beyond the version barrier between sister namespaces if the type of that instance is compatible between these namespaces. The main purpose of this paper is to provide a mechanism for relaxing the version barrier while keeping type-safe instance accesses with negligible performance penalties in regular execution. The mechanism of sister namespaces is implemented by extending the type checker and the class loader of the JVM.

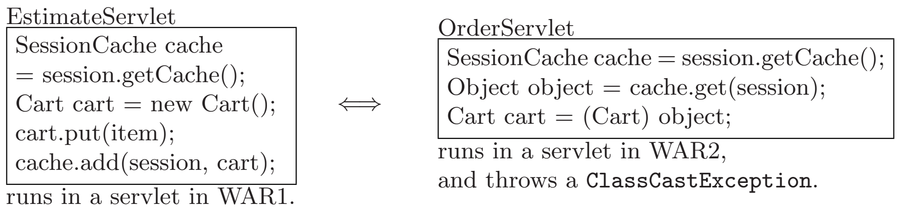
The rest of this paper is organized as follows. Section 2 describes two problems that cause trouble for component-based application developers. Section 3 presents the design and implementation of the sister namespace. Section 4 discusses a few implementation issues. Section 5 presents the results of our experiments. Section 6 compares the sister namespace mechanism to other related work. Section 7 concludes this paper.

2 Problems of the Version Barrier

This section presents two problems that developers often encounter when developing a component-based application in Java. These problems are actually caused by the version barrier between namespaces.

2.1 J2EE Components

Most J2EE platforms, either commercial (e.g., Websphere, Weblogic) or open-source (e.g., JBoss, Tomcat), support both the development and the deployment



EstimateServlet

```
SessionCache cache
= session.getCache();
Cart cart = new Cart();
cart.put(item);
cache.add(session, cart);
```

runs in a servlet in WAR1.

$\Longleftrightarrow$

OrderServlet

```
SessionCache cache = session.getCache();
Object object = cache.get(session);
Cart cart = (Cart) object;
```

runs in a servlet in WAR2,
and throws a `ClassCastException`.

Fig. 1. Passing the session cache from one WAR component to another

of pluggable component archives (EJB-JARs, WARs, and EARs). A Web Application Archive (WAR file) is used to deploy a web-based application. This file can contain servlets, HTML files, Java Server Pages (JSPs), and all associated images and resource files. An Enterprise Application Archive (EAR file) may contain one or more Enterprise JavaBeans (EJBs) and WARs. The functionality of the so-called hot deployment enables such J2EE components to be plugged and unplugged at runtime without restarting the application servers. Thus, a J2EE application can be dynamically customized on a per-component basis. This dramatically improves the productivity of software development. For enabling hot deployment, each component joins a distinct namespace, loaded by a distinct class loader.

However, the version barrier makes it impossible to pass instances of each version of a class across the boundary of J2EE components, or namespaces. Such instances are typically caches, cookies, or session objects or beans. For example, consider the following scenario. An instance of the `Cart` class must be passed between servlets included in different web application archives (that is, from the `EstimateServlet` included in one web archive, WAR1, to the `OrderServlet` in another web archive, WAR2). The class file of the `Cart` class is packaged into a Java Archive (JAR) file, and identical copies of that JAR file reside in the `WEB-INF/lib` directories in each web archive. Thus, each class loader loads the `Cart` class separately. Figure 1 illustrates the implementation of these servlets: `EstimateServlet` puts an instance of `Cart` into the session cache and `OrderServlet` pulls that instance out of the cache. When casting it from `Object` to `Cart`, the JVM will throw a `ClassCastException`. Since the `Cart` class referenced by the `EstimateServlet` class is a distinct type from the type referenced by the `OrderServlet` class, the version barrier prevents assignment of that instance to the variable `cart` in the `OrderServlet` class by throwing a cast error in advance.

Some readers might think the delegation model of class loaders in Java is a solution to the problem above. These WAR components can share the same version of a class if they delegate the loading of that class to their common parent, such as the `EAR` class loader (Figure 2). In fact, the typical J2EE platform has such a common parent loader. Child class loaders can have their parent loader load a class if they want to share the same version of that class. In the case of J2EE, the `SystemClassLoader` is the parent of all EAR class loaders and an EAR class loader is, in turn, the parent of all WAR class loaders included

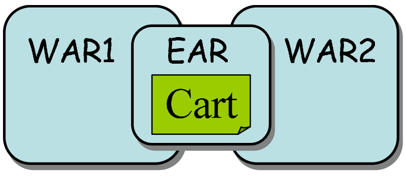


Fig. 2. A parent EAR class loader is used for sharing class types between WAR1 and WAR2. The rounded box represents a namespace for the J2EE component. The overlapping part means the overlapped namespace

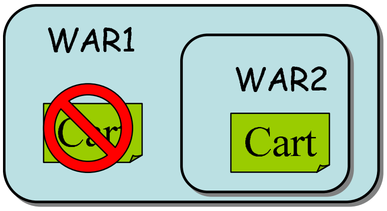


Fig. 3. The JBoss application server based on the unified class loader architecture makes a parent-child relationship between the communicating components

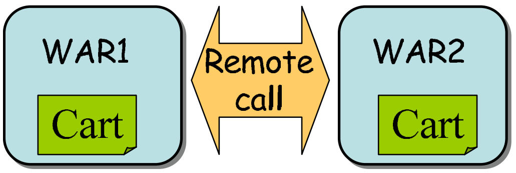


Fig. 4. All inter-component communications are realized by a remote call

in that EAR. However, the solution using a parent class loader tightly couples several irrelevant J2EE components together. Such coarse-grained composition decreases the maintainability and availability of all related software components. For example, consider the two components `DVDStore` and `Pizzeria`: the former models an online 24-hour DVD store and the latter models an online home delivery pizzeria available from hours 10 to 21. If both of these components share the abovementioned application component including `Cart` and if this component is packaged into `Pizzeria`, then undeploying `Pizzeria` for maintenance stops the service by `DVDStore`. Since `DVDStore` must run 24 hours a day, it is almost impossible to decide the maintenance schedule of `Pizzeria`.

To solve this problem, the JBoss application server provides the unified class loader (UCL) architecture [21] for sharing across components across the J2EE components. A collection of UCLs acts as a single class loader, which places into a single namespace all the classes to be loaded. All classes are loaded into the shared repository and managed by this repository. However, this architecture disables different J2EE components with the same name (Figure 3).

Another technique, considered a last resort, is using the Java Serialization API to exchange objects between different J2EE components through a byte stream, which is the referred to as *Call-by-Value* (Figure 4). Typical J2EE platforms adopt this approach for inter-EAR communications. However, even if an EAR wants to transfer an object to another EAR deployed in the same container (or JVM), it must execute a remote call. This remote call is a waste of I/O resources and it decreases the overall performance. Although the Local Interface mechanism introduced in EJB2.0 allows communications between components without remote calls (*Call-by-Reference*), these components must be packaged together in the same archive.

2.2 Eclipse Plug-in Framework

The Eclipse platform [37], an integrated development environment for Java, can be considered as a component system due to its advanced plug-in framework. A plug-in module can contain all sorts of resources, including code and documentation. A plug-in module must also contain sufficient information for the platform to incorporate the code or documentation into itself. The plug-in framework allows us to easily add, update and remove discrete portions of the contents. In addition, since a separate class loader (called a plug-in class loader) is created for each plug-in module, each plug-in module has its unique namespace and is dynamically deployable.

However, the Eclipse plug-in framework has a structural problem due to the version barrier. For example, consider the Eclipse help system plug-in module [12]. It is a useful plug-in module that allows users to develop and deploy professional-quality, easy-to-use, and searchable online documentation. The Eclipse help system can be used as an infocenter, which is an application implemented as a web component and accessible from a web browser. However, to be used as an infocenter, the current Eclipse help system needs to run on a separate process from the process of the web server (Figure 5). The web server must make new processes for the help system and the minimum Eclipse system, and then the web server must dispatch all requests to the help system. Thus, every communication for dispatching requests from the web server to the help system is a remote call, which involves marshalling all passed instances.

A real problem of the example above is that, no matter which namespace the help system joins, all instances must be marshaled and unmarshaled with performance penalties to avoid trouble due to the version barrier when they are passed between the web server and the help system. This is true even if the help system is run on the same process as the web server. Suppose that the

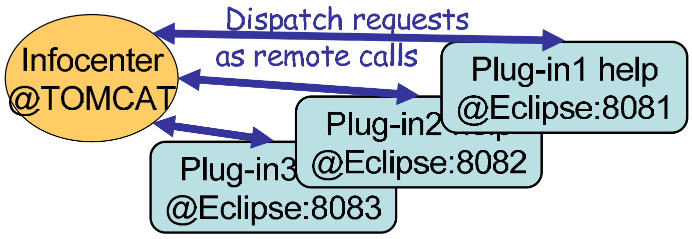


Fig. 5. The Eclipse help system must run as a separate process

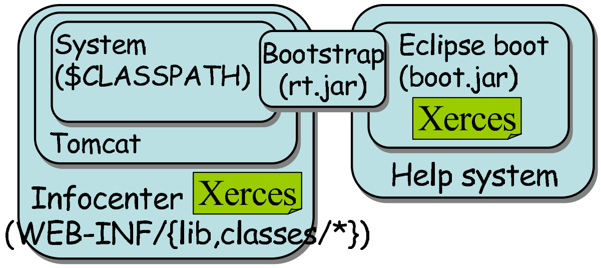


Fig. 6. The Xerces archives are loaded in duplicate for the Eclipse help system and the infocenter

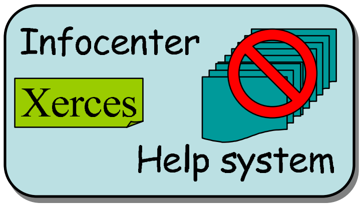


Fig. 7. Loading all components by a class loader breaks the isolation of each namespace

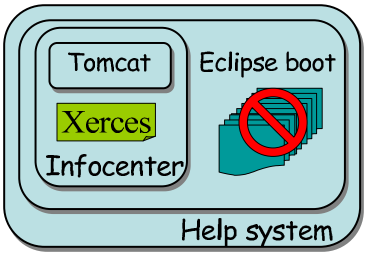


Fig. 8. Delegating the Xerces archives to the web component class loader breaks the isolation of the help system

help system runs on the same JVM as the infocenter, and both the help system and the infocenter use the Apache Xerces [36] archive, which contains an XML parser in the `WEB-INF/lib` directory. If the help system joins a namespace independent of the namespace of the infocenter (Figure 6), the version barrier does not allow the instances of an XML parse tree to be exchanged between the help system and the infocenter, since the copies of the Xerces archive are loaded in duplicate and then different versions of the tree-node class types are created for each archive. If the help system joins the same namespace as the infocenter by deploying as a WAR file into the `WEB-INF/lib` directory (Figure 7), the XML parse tree can be exchanged between the two components. However, this obviously breaks the isolation of the help system from the infocenter. For example, several core components of the Eclipse platform must also be loaded together with the help system, and these core components cause naming conflicts with the infocenter. Furthermore, all the components must be redeployed together when some of the components are redeployed for maintenance. Finally, if the help system joins a descendant namespace of the infocenter (Figure 8), delegating the Xerces archives to the parent class loader also allows sharing the Xerces archives. However, it ends up breaking separated namespaces, too.

2.3 Extending Assignment Compatibility

The problems illustrated above can be solved if the algorithm for computing *assignment compatibility* in the Java programming language is extended to include version conversions between different versions of a class type. Here, the version conversion means a conversion from a version of a class type to any other version of that class type. If this conversion is chosen in the context of assignment, casting, and method invocation conversions such as widening and narrowing conversions, instances could be easily passed across the version barrier [2]. For example, this extension of assignment compatibility would allow assignments between different versions of a class type. Thus, a component would be able to pass instances into and from another component, even if both components load and define that class type separately. The `OrderServlet` class in Figure 1 would not throw a cast error. Moreover, the Eclipse platform would not need to care about where and how many Xerces libraries are available in the current execution environment.

However, naively relaxing the version barrier by extending the assignment compatibility causes a serious security problem. For example, a program may access a non-existing field or method and then crash the JVM. In fact, the version barrier of Sun JDK 1.1 was wrongly relaxed, and thus it had a security hole known as the type-spoofing problem, first reported by Saraswat [26]. This security hole had been solved by the loader constraint scheme [18], which rather strengthens the version barrier. To avoid this security problem while relaxing the version barrier, it would be necessary to have runtime type checking, as is found in dynamically typed languages such as CLOS, Self, and Smalltalk. In such languages, since a variable is not statically typed, any type of instance can be assigned to it. For security, several interpreters for dynamically typed languages perform runtime type checks, called guard tests, so that an exception can be thrown at runtime if a non-existing method or field is accessed. A drawback of this approach is that it requires frequent runtime type checks, which implies non-negligible performance degradation, whereas the JVM performs these runtime type checks. Another technique is to perform runtime type checks at every assignment operation, such as the `aastore` Java bytecode instruction, which is used for storing an object reference in an array object. This operation verifies that the stored object is type-safe. However, this approach also causes performance degradation, since the JVM must perform a type-check for not only `aastore` but also for a large number of other assignment instructions.

3 Sister Namespaces

We propose *sister namespaces*, which can relax the version barrier between namespaces. Different versions of a class type that join sister namespaces can be

[2] If two class types have assignment compatibility with each other, one type can be converted to the other type in the context of not only assignment conversions but also casting and method invocation conversions.

assignment compatible with each other if these versions have differences while still preserving the *version compatibility*. Our challenge is to relax the version barrier while keeping type-safe instance accesses efficient. In this section, we first define extended assignment compatibility, which is based on Java binary compatibility [7] (Section 3.1). Next, we show the sister-supported type checker, which takes the central role in relaxing the version barrier for sister namespaces (Section 3.3). The type checker blocks illegal objects when they move across the version barrier, and thus no subsequent extra check is needed for these objects. This is enabled because it is prohibited for a namespace to become a sister of its parent or child namespace. In addition, we present the sister loader constraint (Section 3.4) and then the schema class loading scheme (Section 3.5). They prevent eager class loading and type inconsistencies, respectively.

We implemented the sister namespaces on the IBM Jikes Research Virtual Machine (JRVM) [1]. The extensions to the JRVM are only the sister-namespace API, a sister-supported class loader, and a sister-supported type checker. The API is provided as an extension to the existing `java.lang.ClassLoader` in the GNU Classpath libraries. These extensions consist of several core classes of the JRVM such as class and object representations.

3.1 Version Compatibility

This section provides the definition of version compatibility, which securely extends the assignment compatibility between different versions of a class type. We define two versions of a class type, C_{ver1} and C_{ver2}, as assignment compatible with each other if C_{ver1} is version compatible with C_{ver2} and vice versa. A class type C_{ver2} is version compatible with C_{ver1} if all the class types that could previously link with C_{ver1} and work with an instance of C_{ver1} without errors are able to also correctly work with instances of C_{ver2} without other assignment compatibility rules such as a subtyping relation. Thus, if C_{ver1} and C_{ver2} are version compatible, then an instance of C_{ver1} can be securely converted to the type C_{ver2} when it is assigned to a variable of C_{ver2} and vice versa. Here, being secure means that every operation on C_{ver2} is applicable to the instance of C_{ver1} without errors; any method call, field access, or type casting applied to the variable does not fail.

The following are the differences that programmers are permitted to make between two versions of a class while preserving version compatibility between the two versions:

– Differences of declared static members such as a static field, a static method, a constructor, or an initializer.
– Differences of the implementation of instance members, such as an instance method.

The differences are derived from the study of the binary compatible changes mentioned in the Java language specification [11].

Version compatibility is based on the idea of binary compatibility; it means that an instance rather than a class can work with the binary of another version of the class type. Java binary compatibility defines a set of changes that developers are permitted to make to a package or to a class or interface type while preserving the compatibility with the preexisting binaries. A change to a class type is binary compatible with preexisting binaries if preexisting binaries that previously linked without errors will continue to link without recompiling. Version compatibility defines differences between two versions of a class type that preserve the binary compatible property between an instance of one version and the binary of the other version. Unlike the original binary compatibility, the version compatibility allows any change to static members since static member accesses are irrelevant to instances. Version compatibility deals with the compatibility between an instance and the binary of another version of that class type. Therefore, to be version compatible, two versions of a class type must have the same set of private members, although the implementations of those members may differ. This is a difference from the binary compatibility, which allows the two versions to have a different set of private members. Since a private member can be accessed from not only `this` instance but also from other instances of another version of that class type, version compatibility requires that the two versions have the same set of private members.

3.2 Creating Sister Namespaces

A sister namespace is a first-class entity, but it is created implicitly when a class loader is instantiated with a class loader given as a parameter. The `ClassLoader` class provides the new constructor as follows:

```
protected ClassLoader(ClassLoader parent, ClassLoader sister)
```

The class loader obtained from this constructor becomes a *sister class loader* of the class loader specified by the parameter `sister`. The latter class loader also becomes a sister of the former one. These two sister class loaders construct their own sister namespaces; the version barrier between them is relaxed if the version compatibility is satisfied. The sister class loaders must not have a parent-child relationship. This rule is significant for the efficient type checking we describe later. If the sister class loaders have such a relationship, the construction of the sister namespaces fails. In this paper, if a version of a class type is loaded earlier (or later) than other versions, it is called a *younger* (or *older*) sister version of that class type. This young-old relationship is independent of the creation order of the sister class loaders.

In the case described in Section 2.1, the application programmers can exchange instances between two namespaces for WAR1 and WAR2 if they are sister namespaces. Each namespace can contain a different version of the type of the exchanged instance. WAR1 and WAR2 must be loaded by the class loaders created as follows:

```
ClassLoader ear = new EARClassLoader();
ClassLoader war1 = new WARClassLoader(ear);
ClassLoader war2 = new WARClassLoader(ear, war1);
```

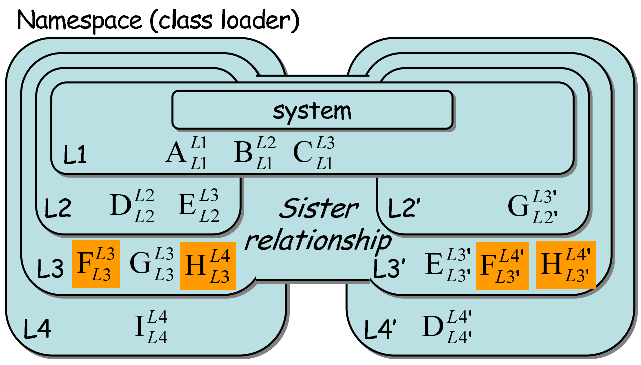

Namespace (class loader)
system
L1 A^{L1}_{L1} B^{L2}_{L1} C^{L3}_{L1}
L2 D^{L2}_{L2} E^{L3}_{L2}
Sister relationship
L2' $G^{L3'}_{L2'}$
L3 F^{L3}_{L3} G^{L3}_{L3} H^{L4}_{L3}
L3' $E^{L3'}_{L3'}$ $F^{L4'}_{L3'}$ $H^{L4'}_{L3'}$
L4 I^{L4}_{L4}
L4' $D^{L4'}_{L4'}$

Fig. 9. The notation $C^{L_i}_{L_d}$ represents a class type, where C denotes the name of the class, L_d denotes the class's defining loader, and L_i denotes the loader initiated class loading. An inclusion relation represents a parent-child relationship. For example, the class loader L1 is a parent of both L2 and L2'. And the classes A, B, C, and the system classes are visible in the namespaces L2, L2', L3, L3', L4, and L4'. In this figure, the sister namespace L3 and L3' have a sister relationship

The `ear`, `war1`, and `war2` are instances of the `ClassLoader` class. The third `new` operation creates sister namespaces for WAR1 and WAR2. Both `war1` and `war2` have the same parent class loader `ear`. In general, application programmers of components, such as Applets, Servlets, Eclipse plug-ins, and EJB, do not have to be aware of namespaces or class loaders. These are implicitly managed by the application middleware. Creating sister namespaces by using the `ClassLoader` constructor above is the work of middleware developers. A sister namespace can make another plain namespace its sister on demand. Since the sister relationship is transitive, if a namespace becomes a sister of a namespace and then it becomes a sister of another namespace, all three namespaces become sisters of each other. Programmers can incrementally create a new namespace and make it another sister of the other sister namespaces. This feature would be useful in cases of incremental development processes and routine maintenance work.

Note that all class types *defined* by a sister class loader can be version compatible with the corresponding sister version of that class type, even if the loading of these class types are *initiated* by the child class loaders. An *initiating* class loader, which initiates the loading of a class type, does not have to actually load a class file. Instead, it can delegate to the parent class loader. The class loader that actually loads a class file and defines that type is called a *defining* class loader of that type. This delegation mechanism is used for sharing the same version of class type between the initiating and defining class loaders. In Figure 9, if two sister namespaces are created between class loaders L3 and L3', the classes F and H can be version compatible with F' and H', respectively. The pairs E and E' or G and G' are not compatible with each other since they are defined by other class loaders.

3.3 Sister-Supported Type Checking

The version barrier is relaxed by a type checker that considers the sister namespaces. In Java programs, most bytecode instructions such as the method invocation instructions `invokevirtual` and `invokenonvirtual`, and field access instructions such as `getfield` and `putfield` are statically typed. These instructions do not perform dynamic type checking. Therefore, these instructions *as they are* can work correctly with any version of class type if they are version compatible. On the other hand, several instructions such as `instanceof`, `checkcast`, `invokeinterface`, `athrow`, and `aastore` entail dynamic type checking. The type checking by those instructions must be enhanced if the version barrier is relaxed so that version compatible instances can be passed between sister namespaces. The algorithm of enhanced type checking for sister namespaces is shown in Figure 10. After the regular type checks are performed, and if they fail (line 2), the extra checks are executed (lines 3–6). First, a sister relationship is examined (line 3). If the left-hand side class type (`LHS`) and the right-hand side class type (`RHS`) have a sister relationship, then the type checker determines whether one class type has undergone the schema compatible loading process against the other type (line 4). Schema compatible loading is introduced later.

Note that these extra checks for sister namespaces are executed only after the regular type checks fail. Since typical programs do not frequently cause type errors, this enhancement for the built-in type checker implies no performance penalties as long as instances are not passed between sister namespaces.

The sister-supported type checker only prohibits a version incompatible instance from being passed between sister namespaces. A version incompatible class can join each of the sister namespaces if an instance of that version stays within the namespace. To avoid the security problem described in Section 2.3 (naively relaxing the assignment compatibility), sister namespaces must detect a version incompatible instance being passed between sister namespaces. This detection is executed by only the `checkcast` instruction. In other words, the detection is not executed by other instructions for method invocation, field access,

```
1: if LHS is a subtype of RHS then true
2: else if LHS is not a subtype of RHS then
3:    if LHS is a sister type of RHS &&
4:       LHS is version compatible with RHS then true
5:    else false
6:    end
7: end
```

Fig. 10. Pseudo code for enhanced type checking for sister namespaces. A type check is the determination of whether a value of one type, hereafter the right-hand side (RHS) type , can legally be converted to a variable of a second type, hereafter the left-hand side (LHS) type. If so, the RHS type is said to be a subtype of the LHS type and the LHS type is said to be a supertype of the RHS type

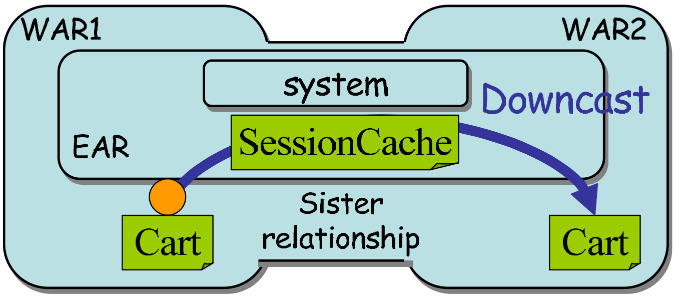


Fig. 11. Downcast enforced by the bridge-safety property satisfied between namespaces

and assignment. This is mainly due to the design of sister namespaces, which must not have a parent-child relationship between them. This rule brings the *bridge-safety* [26] property to all classes included in the sister namespaces. This property guarantees that an instance of a class type is always examined by the `checkcast` instruction when it is passed between sister namespaces. It must be first upcast to a type loaded by the common parent class loader of the two sister class loaders, and then it must be downcast before it is assigned to the class type loaded by the sister class loader at the destination. For example, when an instance of `Cart` is passed, it will be first upcast to a super class of `Cart`, such as the `Object` class, and then downcast to another version of the `Cart` class (Figure 11). Therefore, the `checkcast` instruction is always executed when the instance is downcast to `Cart`.

To implement the sister-supported type check, we modified the `VM_DynamicTypeCheck` class in the JRVM. We extended that class and the TIB (Type Information Block) for fast type checking to consider sister relationships. The original TIB holds several arrays of type identifiers. For example, the arrays of extended superclass types and of implemented interface types are stored in the TIB for fast type checking without looking up the whole type hierarchy [3][2]. Similarly, the extended TIB holds two arrays of `sid`s. The `sid` is the identifier of a sister relationship. The two arrays are of the `sid`s of the extended superclasses and the `sid`s of the implemented interfaces. The `sid` of a class can be obtained from a `VM_Class` object representing that class. We extended the `VM_Class` class to hold the `sid` of the class.

3.4 Sister Loader Constraint

A straightforward implementation of the sister-supported type checker requires eager class loading. Even if the sister-supported type checker verifies that the type of an instance is version compatible, that instance cannot be fully *trusted*. The instance may contain a version incompatible instance as a field value or return it as a result of a method execution. That is, the *untrusted* instance may *relay* an incompatible instance. Since an instance is type checked only when it is downcast, the types of the instance that may be relayed must also be type checked at the same time. Therefore, the type checker verifies all the class types occurring in the class definition of that instance, such as parameter types[3], return types, and field types. It also recursively verifies the class types occurring in the definitions of those types. However, if this recursive type check is naively implemented, all the related classes would have to be eagerly loaded. This eager loading is practically unacceptable, since the advantages of the dynamic features of Java would be lost. The sister-supported type checker must be able to work with the scheme of lazy class loading. Note that the original class loading mechanism of Java is based on lazy class loading.

To examine version compatibility while enabling lazy class loading, the JVM maintains a set of *sister loader constraints*, which are dynamically updated when the sister-supported type checker works. If the type checker finds a class type that must be verified but has not been loaded yet, the JVM does not eagerly load that class; instead, it records a sister loader constraint. For example, if the type checker attempts to verify that a version of class C is version compatible with another version C', but C or C' has not been loaded yet, the JVM records as a constraint that C must be version compatible with C'. This constraint is later verified when C or C' is loaded. If the type checker detects that this constraint is not satisfied, it throws a `LinkageError`. While the type checker is verifying that constraint, if it finds another class type that must be verified but is not loaded, a new sister loader constraint is recorded. If the type checker finds a class type that must be verified and has been already loaded, it recursively verifies that class type at the same time. Note that every constraint is verified only once. The result of the verification is recorded to avoid further verification.

In summary, the JVM needs to maintain the invariant: *Each class type co-existing in the namespace satisfies all the sister loader constraints.* The invariant is maintained as follows:

Every time a new class joins a sister namespace, the JVM verifies whether that class type will violate an existing sister loader constraint.

If the class type being loaded violates an existing sister loader constraint, loading that class type fails since that class type is untrusted in the namespace. If there is no constraint referring to that class type, the JVM loads that class type, although that class type might be version incompatible. It is verified later when a new constraint referring to that class type is recorded.

Every time a new sister loader constraint is recorded, the JVM verifies whether that constraint is satisfied with the class types that have been already loaded.

If a class type that has already been loaded does not satisfy a newly recorded constraint, loading the class type that starts the type checking process producing the new constraint is untrusted and hence the loading is aborted. If any class types needed for verifying that constraint have not been loaded, the verification

[3] If a parameter type is not version compatible, an incompatible instance may be sent back without type checking to the namespace that has sent the instance of the class type including that parameter type.

is postponed until those classes are loaded. Otherwise, if all the class types needed for the verification have been loaded and the constraint is successfully verified, the constraint is removed from the record.

For efficient verification of constraints, we added an array of flags to `VM_Class`. Each flag indicates whether the version of the class type represented by a `VM_Class` object has been recursively type checked with another sister version. The flag is true only if the two versions of the class type are version compatible and if the type checker has verified that those two versions never *relay* a version incompatible instance. Since there might be multiple sisters, the `VM_Class` object holds an array of the flags, each of which indicates the result of the type check with each sister version. The JVM uses these flags for executing a recursive type check only once.

3.5 Schema Compatible Loading

Even if two versions of a class type satisfy the version compatibility, these instances may have schema incompatibility. This means that the layout of the internal type information blocks (TIBs) may not be identical between the two versions of the class type. The TIB holds fields and function pointers to a corresponding method body. The order of the TIB entries depends on the JVM or compilers; it does not depend on the order of the member declarations in a source file or a class file. Thus, even if two versions of a class type have version compatibility, the layout of the TIBs may not be identical.

The sister-supported class loader guarantees that layouts of the TIBs are identical between two versions of a class type if the class types are version compatible. Since the JVM uses a constant index into the TIB when it accesses a field or a method, the JVM cannot correctly execute the bytecode if the layouts of the TIBs are not identical between compatible versions of the class type. Therefore, when the class loader loads a younger version of a class type, the JVM constructs the TIB of that version of the class so that the layout of the TIB is identical to that of the TIB of an older version of the class. This loading process is called *schema compatible loading*. Note that this process is given up against the incompatible class type that has no binary compatibility with the older sister version of that class type. This result is employed by the JVM to quickly examine whether a class type is trusted or not.

In the JRVM, a TIB is constructed during the execution of the `resolve` method in `VM_Class`. The `resolve` method is invoked during the class resolution process by the `VM_ClassLoader` class, an instance of which represents a class loader. The `resolve` method has been extended to perform the schema compatible loading.

4 Discussion

4.1 Canceling JIT Compilations

Just-In-Time (JIT) compiled code sometimes needs to be canceled since a devirtualized method call does not correctly refer to a method declared in a sister version of the class type. Recent optimizing JIT compilers [15, 32] perform the devirtualization optimization that transforms not only a final and a static method but also a virtual method call to a static method. For a given virtual call, the compiler determines whether or not the call can be devirtualized by analyzing the current class hierarchy. If the method can be devirtualized and its code size is small enough, the compiler inlines the method. Therefore, if the type of an instance is converted to a sister version of that class type, the JVM would continue to invoke the original inlined code instead of the real method of that instance. This is because the JIT compiler does not consider sister namespaces; method bodies might be different among sister versions of the same class type.

To avoid this problem, the JIT compiler must cancel devirtualization when a new sister version of a class type is loaded. Fortunately, most optimizing JIT compilers have an efficient cancellation mechanism for dynamic class loading. Since a whole class hierarchy cannot be statically determined in Java, JIT compilers can dynamically replace [32] or rewrite [15] inlined code. This is performed when a new subclass is loaded and the subclass overrides a method that has not been overridden by the other subclasses. The JIT compiler that supports sister namespaces also cancels devirtualization when version compatibility is verified and a new sister version of a class type is available.

4.2 Eager Notifications of Version Incompatibility

Version incompatibility checks between sister versions of a class type may eagerly throw a cast error before any incompatible class types actually co-exist in one namespace. For example, if the type checker detects that a class type may relay a version incompatible instance, it throws a cast error. This eager notification strategy is similar to the loader constraint scheme [18]. The JVM prohibits different versions of a class type from even being loaded if the JVM encounters an operation that relays instances from one namespace to the other. If the JVM has already loaded these versions of the class, the JVM throws a link error. However, except for the loader constraint scheme, verification of compatibility and error notification are not performed at loading time but are done later by the linker, while the linker resolves the constant pool items (e.g., `NoSuchMethodError`, `IllegalAccessError`, `IncompatibleClasChangeError`). If a Java program includes binary incompatibility, it continues to run until it actually executes an illegal operation caused by that incompatibility. This lazy verification and notification are useful in practice.

However, we have adopted the eager strategy for avoiding performance penalties due to version compatibility checks. To delay notifications of incompatibility as long as the program continues to run without errors, a number of guard tests must be embedded into the incompatible class. In Java, once the JVM executes a method invocation or a field access, the operation is linked with the call site and replaced with efficient code that does not perform type checking anymore. Therefore, the guard tests must be embedded for verifying version compatibility after the version incompatibility is found. It requires the JIT compiler to recompile the code that refers the incompatible class type. Moreover, the guard tests imply the non-negligible performance overhead mentioned in Section 2.3; thus, we do not delay the notifications of incompatibility.

5 Experimental Results

This section reports the results of our performance measurements. We performed all the experiments on the IBM Jikes Research Virtual Machine 2.3.2 with Linux kernel 2.4.25, which were running on a Pentium4 1.9GHz processor with 1GB memory. Both the Jikes RVM and our modified RVM were compiled to use the baseline compiler for building the boot image with the semi-space garbage collector.

Baseline Performance. To measure the baseline performance, we ran the SPECjvm98 [31] benchmarks on both our JVM and the unmodified JVM. The problem sizes of all the benchmarks were 100 (maximum). Table 1 lists the results. The numbers are the average execution time for 20 repetitions. The baseline overhead due to the sister namespace was negligible.

Cost of Loading Classes Into Sister Namespaces. We measured the time for loading classes with a plain class loader or a sister class loader. This experiment shows the performance penalty incurred by the sister class loader, which executes schema compatible loading and verifies the version compatibility between classes. We took nine application programs, listed in Table 2, to measure the total loading time. The loading process includes delegating to the parent class loader, searching for a class file in a specified classpath, and resolving, initializing, and instantiating that class type in the JVM. All loading processes are iterated 20 times. The results show that the performance penalty varied among those applications from around 14% to 67%. The penalties mostly depended on the number of declared methods and fields. Thus, the largest application showed the largest overhead.

Table 1. SPECjvm98 benchmark results on both our JVM and the unmodified JVM

Benchmark Program	*Jikes RVM (JRVM)*	*Sister-supported JRVM (SVM)*	*SVM /JRVM*
_201_compress	47.293 ms	46.218 ms	97.7%
_202_jess	40.258 ms	38.726 ms	96.2%
_205_raytrace	22.704 ms	23.404 ms	103.1%
_209_db	65.628 ms	67.075 ms	102.2%
_213_javac	54.698 ms	57.759 ms	105.6%
_222_mpegaudio	29.344 ms	29.210 ms	99.5%
_227_mtrt	25.812 ms	24.563 ms	95.2%
_228_jack	28.372 ms	28.047 ms	98.9%
Total	314.109 ms	315.002 ms	100.3%

Table 2. Total loading time using an ordinary class loader and a sister class loader. All classes are sequentially loaded by the `loadClass()` method

Program		*Total loading time*		*sister*
(No. of classes)		*plain namespace*	*sister namespace*	*/plain*
JDOM	(72 classes)	328 ms	382 ms	116.5%
Crimson	(144 classes)	569 ms	696 ms	122.3%
jaxen	(191 classes)	802 ms	919 ms	114.6%
dom4j	(195 classes)	1,308 ms	1,487 ms	113.7%
SAXON	(351 classes)	1,749 ms	2,113 ms	120.8%
XT	(466 classes)	1,223 ms	1,422 ms	116.3%
XercesJ 1	(579 classes)	2,495 ms	3,046 ms	122.1%
XercesJ 2	(991 classes)	4,144 ms	6,177 ms	149.1%
XalanJ 2	(1,548 classes)	12,884 ms	15,290 ms	166.6%

JDOM [38] : A simple Java representation of an XML document, version 1.0
Crimson [34] : A Java XML parser included with JDK 1.4 and greater, version 1.1.3
jaxen [39] : An XPath engine, version 1.0.
dom4j [23] : The flexible xml framework for Java, version 1.5.2
SAXON [17] : An XSLT and XQuery processor, version 6.5.3
XT [19] : A fast, free implementation of XSLT in java, version 20020426a
XercesJ 1 : The Xerces Java Parser 1.4.4.
XercesJ 2 : The Xerces2 Java Parser 2.6.2.
XalanJ 2 [35] : An XSLT processor for transforming XML documents, version 2.6.0.

Cost of the `Checkcast` Instruction. Finally, we measured the execution time for type checking. The sister-supported type checking includes not only the ordinary `checkcast` operation but also the checking of trusted instances. We ran a program that executes the `checkcast` instruction for every class included in a given application, and then we measured the total execution time of all the `checkcast` instructions. Both experiment programs ran after all the classes had been loaded and then the version compatibility of all the classes was verified. We successively ran the program twice; the execution time of the second run indicates the execution time of `checkcast` after the version compatibility of all the possibly relayed classes is verified during the first run. Some of the first checks also make use of the results of previous verifications.

Table 3 lists the results. The results are the average of 10,000 iterations. The total execution time of the first checks was from about 10 to 40 times slower than the ordinary `checkcast` operation. This is because the sister-supported type checker traverses all possibly relayed class types. Since each application has a different number of possibly relayed classes, the relative performance varies for each application. On the other hand, the second checks included only around 160% overhead compared to the ordinary `checkcast` operation. Note that this overhead is incurred only when `checkcast` examines the type of an instance coming from another sister namespace. The overhead is negligible in regular cases.

We also compared the execution time of the type check with the time for marshalling and unmarshalling several XML data objects. Remember that the most harmless practice for the inter-component communication described in Section 2

Table 3. Total execution time of the type check by `checkcast`

Program	*(No. of classes)*	`checkcast`	*Sister namespaces*		*Relative performance*	
			first	*second*	*first*	*second*
JDOM	(72 classes)	33.3 us	1,205.7 us	53.7 us	3,721%	261.3%
Crimson	(144 classes)	69.0 us	1,659.7 us	112.6 us	2,505%	263.1%
jaxen	(191 classes)	89.2 us	1,573.1 us	139.8 us	1,864%	256.7%
dom4j	(195 classes)	109.7 us	4,371.7 us	185.8 us	4,085%	269.3%
SAXON	(351 classes)	295.7 us	5,141.3 us	499.8 us	1,839%	269.0%
XT	(466 classes)	381.5 us	4,505.8 us	698.7 us	1,281%	283.1%
XercesJ 1	(579 classes)	644.4 us	7,824.3 us	1,041.3 us	1,314%	261.5%
XercesJ 2	(991 classes)	1,158.6 us	11,534.6 us	1,798.0 us	1,096%	255.1%
XalanJ 2	(1,548 classes)	1,650.6 us	22,627.8 us	2,696.2 us	1,471%	263.3%

is using a remote call, which passes an object by means of the call-by-value. This practice lets us avoid the problem of the version barrier, but it implies overhead due to the marshalling and unmarshalling for parameter passing. On the other hand, sister namespaces also let us avoid the problem, and it implies extra overhead only due to the type check. We measured the execution time for marshalling and unmarshalling DOM objects created by XercesJ 2.6.2 from 33 XML files taken from the Eclipse help system, which includes the Platform, Workbench, JDT, Plug-in and PDE document plug-ins. The overall file size was about 400KB. The measured execution time was 3 million times larger than the execution time of the ordinary `checkcast` operation. Of course, actual inter-component communication using a remote call would spend much more time for the network data transportations. Therefore, this result shows the sister namespace is a significantly faster solution compared to the solution of passing objects by a remote call.

6 Related Work

In the object database community, several schema evolution techniques such as schema or class versioning [5, 30] have been studied. These techniques allow multiple co-existing versions of a schema or a class. Instances are evolved when passing through the version barrier into the modified application or other applications. Using such evolvable object databases is a workable alternative for component-based applications. However, our work concentrates on programming environments, especially where runtime overheads due to schema evolution must be severely minimized.

There have been other research activities tackling the version barrier problem in the programming language and environment community. Most of the previous research regarded the version barrier as a temporal boundary between *old* and *new* components and thus focused on dynamic software updates or evolution. That is, multiple versions of the same class type could not simultaneously co-exist in the running program. Therefore, our problems were not directly addressed. In this research area, the main topic is which existing object should be adapted to an updated version of the class type and how and when. For example, the work on the hotswapping of classes falls into this category. Malabarba *et al.* [20] modified the JVM to make a class reloadable at runtime so that all existing objects can be updated incrementally. The JPDA (Java Platform Debugger Architecture) [33] and the `java.lang.instrument` package of the Java2 SDK5.0 provide the restricted hotswap functionality, whereby existing instances can be considered as the new version of the class type without being updated. Hjalmtysson and Gray [13] implemented dynamic classes in C++ by using templates. Users can selectively update some but not all objects with the help of wrapper (or proxy) classes and methods. Hnětynka *et al.* [14] proposed the renaming approach using a bytecode manipulation tool. A class loader using this renaming approach allows the reloading of a class, although it renames that class.

Our work mainly focuses on the spatial version barriers among multiple components. An older version of a class type remains after a new version is loaded. Dynamic type changes such as predicate classes [4], reclassifying objects [6], and wide classes [28] may allow relaxing of the spatial version barrier, since multiple class members can be implicitly merged by the explicit composition operation. Type-based hotswapping proposed by Duggan *et al.* [8] is similar to our work but classified into the same category as the above systems. The .NET counterparts of the Java class loaders are *application domains*, which are used to load and execute assemblies and can run in a single process. However, they adopt the call-by-value semantics on inter-component communication between application domains using the .NET Remoting Framework. Of course, dynamic typing languages, such as CLOS, Self, and Smalltalk, provide more flexible mechanisms for allowing types to be changed at runtime. However, our challenge is relaxing only the version barrier in the strictly typing object-oriented world with negligible performance penalties. Our contribution is that we have provided a simple mechanism for relaxing the version barrier, which has been confusing Java programmers because of the complicated semantics.

7 Conclusion

This paper presented the design and implementation of loosely-separated sister namespaces in Java. Combining multiple namespaces as sister namespaces can relax the version barrier between them. It thereby allows an instance to be assigned to a different version of that class type in that sister namespace. This mechanism was implemented on the IBM Jikes RVM for evaluation of the performance overhead. Our experiment showed that, once an instance passes into the sister namespace across the version barrier, all instances of that class type can go back and forth between the sister namespaces with significantly low performance overhead. Our experiment also demonstrated that the execution performance has only negligible overhead unless an instance is passed across the version barrier.

We plan to develop a dynamic AOP (Aspect-Oriented Programming) system based on sister namespaces. We have developed a Java-based dynamic AOP system called Wool [27]. It allows weaving aspects with a program at runtime by using the hotswap mechanism of the standard debugger interface called JPDA (Java Platform Debugger Architecture) [33]. The version compatible changes shown in this paper are almost the same as that supported by the JPDA. Therefore, using sister namespaces will make our dynamic AOP system simpler and more efficient while keeping the equivalent flexibility.

Currently, our primary focus for future work is the formal proof of the type safety on the sister namespaces. We will also examine this issue with respect to the Java security architecture [10].

Acknowledgement

We would like to express our deep gratitude to the anonymous reviewers. Hidehiko Masuhara gave helpful comments on an early draft of this paper. We also thank Romain Lenglet for his great efforts to fix numerous English problem in this paper. This research was partly supported by the CREST program of Japan Science and Technology Corp.

References

1. Alpern, B., Attanasio, C.R., Barton, J.J., Burke, M.G., Cheng, P., Choi, J.D., Cocchi, A., Fink, S.J., Grove, D., Hind, M., Hummel, S.F., Lieber, D., Litvinov, V., Mergen, M.F., Ngo, T., Russell, J.R., Sarkar, V., Serrano, M.J., Shepherd, J.C., Smith, S.E., Sreedhar, V.C., Srinivasan, H., Whaley, J.: The Jalapeno virtual machine. IBM System Journal **39** (2000) 211–238
2. Alpern, B., Cocchi, A., Fink, S.J., Grove, D., Lieber, D.: Efficient implementation of java interfaces: Invokeinterface considered harmless. In: Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2001). Number 11 in SIGPLAN Notices, vol.36, Tampa, Florida, USA, ACM (2001) 108–124
3. Alpern, B., Cocchi, A., Grove, D.: Dynamic type checking in jalapeño. In: Java Virtual Machine Research and Technology Symposium. (2001)
4. Chambers, C.: Predicate classes. In: ECOOP'93 - Object-Oriented Programming, 7th European Conference. Volume 707 of Lecture Notes in Computer Science., Kaiserslautern, Germany, Springer-Verlag (1993) 268–296
5. Clamen, S.M.: Type evolution and instance adaptation. Technical Report CMU-CS-92–133, Carnegie Mellon University School of Computer Science, Pittsburgh, PA (1992)
6. Drossopoulou, S., Damiani, F., Dezani-Ciancaglini, M., Giannini, P.: Fickle : Dynamic object re-classification. In: ECOOP 2001 - Object-Oriented Programming, 15th European Conference. Volume 2072 of Lecture Notes in Computer Science., Budapest, Hungary, Springer (2001) 130–149
7. Drossopoulou, S., Wragg, D., Eisenbach, S.: What is java binary compatibility? In: Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98), Vancouver, British Columbia, Canada (1998) 341–361

8. Duggan, D.: Type-based hot swapping of running modules. In: Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01). Volume 10 of SIGPLAN Notices 36., Florence, Italy, ACM (2001) 62–73
9. Englander, R.: Developing Java Bean. O'Reilly and Associates, Inc. (1997)
10. Gong, L., Ellison, G., Dageforde, M.: Inside Java2^{TM} Platform Security: Architecture, API Design, and Implementation 2nd Edition. Addison-Wesley, Boston, Mass. (2003)
11. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification Second Edition. Addison-Wesley, Boston, Mass. (2000)
12. Halsted, K.L., Roberts, J.H.J.: Eclipse help system: an open source user assistance offering. In: Proceedings of the 20st annual international conference on Documentation, SIGDOC 2002, Toronto, Ontario, Canada, ACM (2002) 49–59
13. Hjálmtyśson, G., Gray, R.: Dynamic C++ Classes: A lightweight mechanism to update code in a running program. In: Proceedings of the USENIX Annual Technical Conference, New Orleans, Louisiana, USENIX (1998)
14. Hnětynka, P., Tůma, P.: Fighting class name clashes in java component systems. In: Modular Programming Languages, Joint Modular Languages Conference, JMLC 2003. Volume 2789 of Lecture Notes in Computer Science., Klagenfurt, Austria, Springer (2003) 106–109
15. Ishizaki, K., Kawahito, M., Yasue, T., Komatsu, H., Nakatani, T.: A study of devirtualization techniques for a java$^{\mathrm{tm}}$ just-in-time compiler. In: Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000). Number 10 in SIGPLAN Notices, vol.35, Minneapolis, Minnesota, USA, ACM (2001) 294–310
16. JUnit FAQ: Why do I get an error (ClassCastException or LinkageError) using the GUI TestRunners?, available at: http://junit.sourceforge.net/doc/faq/faq.htm. (2002)
17. Kay, M.: SAXON The XSLT and XQuery Processor, available at: http://saxon.sourceforge.net/. (2001)
18. Liang, S., Bracha, G.: Dynamic Class Loading in the Java Virtual Machine. In: Proceedings of OOPSLA'98, Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications. Number 10 in SIGPLAN Notices, vol.33, Vancouver, British Columbia, Canada, ACM (1998) 36–44
19. Lindsey, B.: XT, available at: http://www.blnz.com/xt/. (2002)
20. Malabarba, S., Pandey, R., Gragg, J., Barr, E., Barnes, J.F.: Runtime Support for Type-Safe Dynamic Java Classes. In: Proceedings of ECOOP 2000 - Object-Oriented Programming, 14th European Conference. Volume 1850 of Lecture Notes in Computer Science., Springer-Verlag (2000) 337–361
21. Marc Fleury, F.R.: The JBoss Extensible Server. In: ACM/IFIP/USENIX International Middleware Conference. Volume 2672 of Lecture Notes in Computer Science., Rio de Janeiro, Brazil, Springer (2003) 344–373
22. Matena, V., Stearns, B.: Applying Enterprise JavaBeans^{TM}: Component-Based Development for the J2EE^{TM} Platform. Pearson Education (2001)
23. Metastaff, Ltd.: dom4j: the flexible xml framework for Java, available at: http://www.dom4j.org/. (2001)
24. Nathan, A.: .NET and COM: The Complete Interoperability Guide. Sams (2002)
25. OMG: The Common Object Request Broker: Architecture and Specification. Revision 2.0. OMG Document (1995)

26. Saraswat, V.: Java is not type-safe. (1997)
27. Sato, Y., Chiba, S., Tatsubori, M.: A Selective, Just-In-Time Aspect Weaver. In: Second International Conference on Generative Programming and Component Engineering (GPCE'03), Erfurt Germany (2003) 189–208
28. Serrano, M.: Wide classes. In: ECCOP'99 - Object-Oriented Programming, 13th European Conference. Volume 1628 of Lecture Notes in Computer Science., Lisbon, Portugal, Springer-Verlag (1999) 391–415
29. Shavor, S., D'Anjou, J., Fairbrother, S., Kehn, D., Kellerman, J., McCarthy, P.: The Java Developer's Guide to Eclipse. Addison-Wesley (2003)
30. Skarra, A.H., Zdonik, S.B.: The management of changing types in an object-oriented database. In: Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86). Volume 11 of SIGPLAN Notices 21., Portland, Oregon (1986) 483–495
31. Spec - The Standard Performance Evaluation Corporation: SPECjvm98. (1998)
32. Sun Microsystems: The Java HotSpot Performance Engine Architecture, available at: http://java.sun.com/products/hotspot/whitepaper.html. (1999)
33. Sun Microsystems: Java^{TM} Platform Debugger Architectuer, available at: http://java.sun.com/j2se/1.4/docs/guide/jpda. (2001)
34. The Apache XML Project: Crimson Java Parser, available at: http://xml.apache.org/crimson. (2000)
35. The Apache XML Project: Xalan Java XSLT Processor, available at: http://xml.apache.org/xalan-j. (2002)
36. The Apache XML Project: Xerces2 Java Parser, available at: http://xml.apache.org/xerces2-j. (2002)
37. The Eclipse Foundation: Eclipse.org, homepage : http://www.eclipse.org/. (2001)
38. The JDOM^{TM} Projec: JDOM, available at: http://www.jdom.org/. (2000)
39. The Werken Company: jaxen: universal java xpath engine, available at: http://jaxen.org/. (2001)

Efficiently Refactoring Java Applications to Use Generic Libraries

Robert Fuhrer[1], Frank Tip[1], Adam Kieżun[2], Julian Dolby[1], and Markus Keller[3]

[1] IBM T.J. Watson Research Center, P.O.Box 704, Yorktown Heights, NY 10598, USA
{rfuhrer, ftip, dolby}@us.ibm.com
[2] MIT Computer Science & AI Lab, 32 Vassar St, Cambridge, MA 02139, USA
akiezun@mit.edu
[3] IBM Research, Oberdorfstrasse 8, CH-8001 Zürich, Switzerland
markus_keller@ch.ibm.com

Abstract. Java 1.5 generics enable the creation of reusable container classes with compiler-enforced type-safe usage. This eliminates the need for potentially unsafe down-casts when retrieving elements from containers. We present a *refactoring* that replaces raw references to generic library classes with parameterized references. The refactoring infers actual type parameters for allocation sites and declarations using an existing framework of type constraints, and removes casts that have been rendered redundant. The refactoring was implemented in Eclipse, a popular open-source development environment for Java, and laid the grounds for a similar refactoring in the forthcoming Eclipse 3.1 release. We evaluated our work by refactoring several Java programs that use the standard collections framework to use Java 1.5's generic version instead. In these benchmarks, on average, 48.6% of the casts are removed, and 91.2% of the compiler warnings related to the use of raw types are eliminated. Our approach distinguishes itself from the state-of-the-art [8] by being more scalable, by its ability to accommodate user-defined subtypes of generic library classes, and by being incorporated in a popular integrated development environment.

1 Introduction

Java 1.5 generics enable the creation of reusable class libraries with compiler-enforced type-safe usage. Generics are particularly useful for building homogeneous collections of elements that can be used in different contexts. Since the element type of each generic collection instance is explicitly specified, the compiler can statically check each access, and the need for potentially unsafe user-supplied downcasts at element retrieval sites is greatly reduced. Java's standard collections framework in package `java.util` undoubtedly provides the most compelling uses of generics. For Java 1.5, this framework was modified to include generic versions of existing container classes[1] such as `Vector`. For example, an application that instantiates `Vector<E>` with, say, `String`, obtaining `Vector<String>`, can only add and retrieve `String`s. In the

[1] For convenience, the word "class" will frequently be used to refer to a class or an interface.

previous, non-generic version of this class, the signatures of access methods such as `Vector.get()` refer to type `Object`, which prevents the compiler from ensuring type-safety of vector operations, and therefore down-casts to `String` are needed to recover the type of retrieved elements. When containers are misused, such downcasts fail at runtime, with `ClassCastExceptions`.

The premise of this research is that, now that generics are available, programmers will want to *refactor* [10] their applications to replace references to non-generic library classes with references to generic versions of those classes, but performing this transformation manually on large applications would be tedious and error-prone [15]. Therefore, we present a refactoring algorithm for determining the actual type parameters with which occurrences of generic library classes can be instantiated[2]. This refactoring rewrites declarations and allocation sites to specify actual type parameters that are inferred by type inference, and removes casts that have been rendered redundant. Program behavior is preserved in the sense that the resulting program is type-correct and the behavior of operations involving run-time types (i.e., method dispatch, casts, and instanceof tests) is preserved. Our approach is applicable to any class library for which a generic equivalent is available, but we will primarily use the standard collections framework to illustrate the approach.

Our algorithm was implemented in Eclipse (see `www.eclipse.org`), a popular open-source integrated development environment (IDE), and parts of this research implementation will be shipped with the forthcoming Eclipse 3.1 release. We evaluated the refactoring on a number of Java programs of up to 90,565 lines, by refactoring these to use Java 1.5's generic container classes. We measured the effectiveness of the refactoring by counting the number of removed downcasts and by measuring the reduction in the number of "unchecked warnings" issued by the Java 1.5 compiler. Such warnings are issued by the compiler upon encountering raw occurrences of generic classes (i.e., references to generic types without explicitly specified actual type parameters). In the benchmarks we analyzed, on average, 48.6% of all casts are removed, and 91.2% of the unchecked warnings are eliminated. Manual inspection of the results revealed that the majority of casts caused by the use of non-generic containers were removed by our refactoring, and that the remaining casts were necessary for other reasons. The refactoring scales well, and takes less than 2 minutes on the largest benchmark.

The precision of our algorithm is comparable to that by Donovan et al. [8], which is the state-of-the-art in the area, but significant differences exist between the two approaches. First, our approach is more scalable because it does not require context-sensitive analysis. Second, our method can infer generic supertypes for user-defined subtypes of generic library classes[3] (e.g., we can infer that a class `MyIterator` extends `Iterator<String>`). The approach of [8] is incapable of making such inferences, and therefore removes fewer casts on several of the benchmarks we analyzed. Third, Donovan et al. employ a strict notion of preserving behavior by demanding that the program's erasure [4] is preserved. In addition to this mode, our tool supports more

[2] This problem is referred to as the instantiation problem in [8].

[3] The version of our refactoring that will be delivered in Eclipse 3.1 will not infer generic supertypes and will always preserve erasure.

relaxed notions of preserving behavior that allow the rewriting of other declarations. Our experiments show that, in some cases, this added flexibility enables the removal of more casts and unchecked warnings. Fourth, our implementation is more practical because it operates on standard Java 1.5 source code, and because it is fully integrated in a popular IDE.

The remainder of the paper is organized as follows. Section 2 overviews the Java 1.5 generics, and Section 3 presents a motivating example to illustrate our refactoring. Sections 4–6 present the algorithm, which consists of the following steps. First, a set of *type constraints* [17] is inferred from the original program's abstract syntax tree (AST) using two sets of generation rules: (i) standard rules that are presented in Section 4 and (ii) generics-related rules that are presented in Section 5. Then, the resulting system of constraints is solved, the program's source code is updated to reflect the inferred actual type parameters, and redundant casts are removed, as discussed in Section 6. Section 7 discusses the implementation of our algorithm in Eclipse, and experimental results are reported in Section 8. We report on experiments with a context-sensitive version of our algorithm in Section 9. Finally, related work and conclusions are discussed in Sections 10 and 11, respectively.

2 Java Generics

This section presents a brief, informal discussion of Java generics. For more details, the reader is referred to the Java Language Specification [4], and to earlier work on the Pizza [16] and GJ [5, 13] languages.

In Java 1.5, a class or interface C may have *formal type parameters* $T_1, \cdots, T_n$ that can be used in non-`static` declarations within C. Type parameter T_j may be *bounded* by types $B_j^1, \cdots, B_j^k$, at most one of which may be a class. Instantiating a generic class $C{<}T_1, \cdots, T_n{>}$ requires that n *actual type parameters* $A_1, \cdots, A_n$ be supplied, where each A_j must satisfy the bounds (if any) of the corresponding formal type parameter T_j. Syntactically, (formal and actual) type parameters follow the class name in a comma-separated list between '$<$' and '$>$', and bounds on formal type parameters are specified using the keyword `extends` (multiple bounds are separated by '`&`'). A class may inherit from a parameterized class, and its formal type parameters may be used as actual type parameters in instantiating its superclass. For example:

```
class B<T1 extends Number>{ ... }
class C<T2 extends Number> extends B<T2>{ ... }
class D extends B<Integer>{ ... }
```

shows: (i) a class `B` that has a formal type parameter `T1` with an upper bound of `Number`, (ii) a class `C` with a formal type parameter `T2` (also bounded by `Number`) that extends `B<T2>`, and (iii) a non-parametric class `D` that extends `B<Integer>`. `B` and `C` can be instantiated with any subtype of `Number` such as `Float`, so one can write:

```
B<Float> x = new C<Float>();
B<Integer> y = new D();
```

Unlike arrays, generic types are *not* covariant: $C{<}A{>}$ is a subtype of $C{<}B{>}$ if and only if $A = B$. Moreover, arrays of generic types are not allowed [14].

Type parameters may also be associated with methods. Such parameters are supplied at the beginning of the generic method's signature, after any qualifiers. For example, a class may declare a generic method as follows:

```
  public <T3> void zap(T3 z){ ... }
```

Calls to generic methods do not need to supply actual type parameters because these can be inferred from context.

Wildcards [21] are unnamed type parameters that can be used in declarations. Wildcards can be bounded from above or below, as in `? extends B`, or `? super B`, respectively. For example, interface `Collection<E>` of the Java 1.5 standard collections library defines a method

```
 boolean addAll(Collection<? extends E> c){ ... }
```

in which the wildcard specifies the "element type" of parameter `c` to be a subtype of formal type parameter `E`, thus permitting one to add a collection of, say, `Floats` to a collection of `Numbers`.

For backward compatibility, one can refer to a generic class without specifying type parameters. Operations on such "raw types" result in compile-time "unchecked warnings"[4] in cases where type-safety cannot be guaranteed (e.g., when calling certain methods on a receiver expression of a raw type). Unchecked warnings indicate the potential for class-cast exceptions at run-time, and the number of such warnings is a rough measure of the potential lack of type safety in the program.

3 Motivating Example

We will use the Java standard collections library in package `java.util` to illustrate our refactoring. In Java 1.5, `Collection` and its subtypes (e.g., `Vector` and `List`) have a type parameter representing the collection's element type, `Map` and its subtypes (e.g., `TreeMap` and `Hashtable`) have two type parameters representing the type of its key and its value, respectively, and `Iterator` has a single type parameter representing the type of object returned by the `next()` method.

Figure 1 shows a Java program making nontrivial use of several kinds of containers. In this program, class `IntList` contains an array of `ints`, and provides an iterator over its elements, and a method for summing its elements. The `iterator()` method creates a `ListIterator`, a local implementation of `Iterator` that returns `Integer` objects wrapping the values stored in an `IntList`. Class `Example`'s `main()` method creates `IntLists` as well as several objects of various standard library types. Executing the example program prints the list `[[2.0, 4.4]]`. The example program illustrates several salient aspects of the use of standard container classes:

– nested containers (here, a `Vector` of `Vectors`), on line 14,
– iterators over standard containers, on line 19,

[4] Unchecked warnings are issued by Sun's javac 1.5 compiler when given the `-Xlint:unchecked` option.

```
public class Example{
  public static void main(String[] args){
    Map m1 = new HashMap();
    Double d1 = new Double(3.3);
    Double d2 = new Double(4.4);
    IntList list1 = new IntList(new int[]{ 16, 17 });
    IntList list2 = new IntList(new int[]{ 18, 19 });
    m1.put(d1, list1); m1.put(d2, list2);
    Vector v1 = new Vector();
    v1.add(new Float(2.0));
    List list5 = new ArrayList();
    list5.add(find(m1, 37));
    v1.addAll(list5);
    Vector v2 = new Vector();
    v2.add(v1);
    System.out.println(v2);
  }
  static Object find(Map m2, int i){
    Iterator it = m2.keySet().iterator();
    while (it.hasNext()){
      Double d3 = (Double)it.next();
      if (((IntList)m2.get(d3)).sum()==i) return d3;
    }
    return null;
  }
}
class IntList{
  IntList(int[] is){ e = is; }
  Iterator iterator(){ return new ListIterator(this); }
  int sum(){ return sum2(0); }
  int sum2(int j){
    return (j==e.length ? 0 : e[j]+sum2(j+1)); }
  int[] e;
}
class ListIterator implements Iterator{
  ListIterator(IntList list3){
    list4 = list3; count = 0; }
  public boolean hasNext(){
    return count+1 < list4.e.length; }
  public Object next(){
    return new Integer(list4.e[count++]); }
  public void remove(){
    throw new UnsupportedOperationException(); }
  private int count;
  private IntList list4;
}
```

Fig. 1. Example program that uses non-generic container classes. Program constructs that give rise to unchecked warnings are indicated using wavy underlining

```
public class Example{
  public static void main(String[] args){
    Map<Double,IntList> m1 = new HashMap<Double,IntList>();
    Double d1 = new Double(3.3);
    Double d2 = new Double(4.4);
    IntList list1 = new IntList(new int[]{ 16, 17 });
    IntList list2 = new IntList(new int[]{ 18, 19 });
    m1.put(d1, list1); m1.put(d2, list2);
    Vector<Number> v1 = new Vector<Number>();
    v1.add(new Float(2.0));
    List<Double> list5 = new ArrayList<Double>();
    list5.add(find(m1, 37));
    v1.addAll(list5);
    Vector<Vector<Number>> v2 = new Vector<Vector<Number>>();
    v2.add(v1);
    System.out.println(v2);
  }
  static Double find(Map<Double,IntList> m2, int i){
    Iterator<Double> it = m2.keySet().iterator();
    while (it.hasNext()){
      Double d3 = it.next();
      if ((m2.get(d3)).sum() == i) return d3;
    }
    return null;
  }
}
class IntList{
  IntList(int[] is){ e = is; }
  ListIterator iterator(){ return new ListIterator(this); }
  int sum(){ return sum2(0); }
  int sum2(int j){
    return (j==e.length ? 0 : e[j]+sum2(j+1)); }
  int[] e;
}
class ListIterator implements Iterator<Integer>{
  ListIterator(IntList list3){
    list4 = list3; count = 0; }
  public boolean hasNext(){
    return count+1 < list4.e.length; }
  public Integer next(){
    return new Integer(list4.e[count++]); }
  public void remove(){
    throw new UnsupportedOperationException(); }
  private int count;
  private IntList list4;
}
```

Fig. 2. Refactored version of the program of Figure 1. Underlining indicates declarations and allocation sites for which a different type is inferred, and expressions from which casts have been removed

– methods like `Collection.addAll()` that combine the contents of containers, on line 13,
– methods like `Map.keySet()` that expose the constituent components of standard containers (namely, a `java.util.Set` containing the `Map`'s keys), on line 19,
– a user-defined subtype (`ListIterator`) of a standard container type, on line 34, and
– the need for down-casts (lines 21 and 22) to recover type information.

Compiling the example program with Sun's javac 1.5 compiler yields six unchecked warnings, which are indicated in Figure 1 using wavy underlining. For example, for the call `m1.put(d1, list1)` on line 8, the following message is produced: "warning: [unchecked] unchecked call to `put(K,V)` as a member of the raw type `java.util.Map`".

Figure 2 shows the result of our refactoring algorithm on the program of Figure 1. Declarations and allocation sites have been rewritten to make use of generic types (on lines 3, 9, 11, 14, 18, and 19), and the down-casts have been removed (on lines 21 and 22). Moreover, note that `ListIterator` (line 34) now implements `Iterator<Integer>` instead of raw `Iterator`, and that the return type of `ListIterator.next()` on line 37 has been changed from `Object` to `Integer`. This latter change illustrates the fact that inferring a precise generic type for declarations and allocation sites may require changing the declared types of non-containers in some cases. The resulting program is type-correct, behaves as before, and compiling it does not produce any unchecked warnings.

4 Type Constraints

This paper extends a model of type constraints [17] previously used by several of the current authors for refactorings for generalization [20] and for the customization of Java container classes [7]. We only summarize the essential details of the type constraints framework here, and refer the reader to [20] for more details.

In the remainder of the paper, $\mathcal{P}$ will denote the original program. Type constraints are generated from $\mathcal{P}$'s abstract syntax tree (AST) in a syntax-directed manner. A set of constraint generation rules generates, for each program construct in $\mathcal{P}$, one or more type constraints that express the relationships that must exist between the declared types of the construct's constituent expressions, in order for that program construct to be type-correct. By definition, a program is *type-correct* if the type constraints for all constructs in that program are satisfied. In the remainder of this paper, we assume that $\mathcal{P}$ is type-correct.

Figure 3 shows the notation used to formulate type constraints. Figure 4 shows the syntax of type constraints[5]. Figure 5 shows constraint generation rules for a number

[5] In this paper, we assume that type information about identifiers and expressions is available from a compiler or type checker. Two syntactically identical identifiers will be represented by the same constraint variable if only if they refer to the same entity. Two syntactically identical expressions will be represented by the same constraint variable if and only if they correspond to the same node in the program's abstract syntax tree.

M, M'	methods (signature, return type, and a reference to the method's declaring class are assumed to be available)
m, m'	method names
F, F'	fields (name, type, and declaring class are assumed to be available)
f, f'	field names
C, C'	classes and interfaces
K, W, V, T	formal type parameters
$E, E', E_1, E_2, \ldots$	expressions (corresponding to a specific node in the program's AST)
$[E]$	the type of expression or declaration element E
$[E]_{\mathcal{P}}$	the type of E in the original program $\mathcal{P}$
$[M]$	the declared return type of method M
$[F]$	the declared type of field F
$\mathit{Decl}(M)$	the class that declares method M
$\mathit{Decl}(F)$	the class that declares field F
$\mathit{Param}(M, i)$	the i-th formal parameter of method M
$T(E)$	actual type parameter T in the type of the expression E
$T(C)$	actual type parameter T of class C
$\mathit{RootDefs}(M)$	$\{\, \mathit{Decl}(M') \mid M$ overrides M', and there exists no $M''(M'' \neq M')$ such that M' overrides $M'' \,\}$

Fig. 3. Notation used for defining type constraints

$\alpha = \alpha'$	type α must be the same as type α'
$\alpha \leq \alpha'$	type α must be the same as, or a subtype of type α'
$\alpha \leq \alpha_1$ **or** $\cdots$ **or** $\alpha \leq \alpha_k$	$\alpha \leq \alpha_i$ must hold for at least one i, $(1 \leq i \leq k)$

Fig. 4. Syntax of type constraints. Constraint variables α, α', ... represent the types associated with program constructs and must be of one of the following forms: (i) a type constant, (ii) the type of an expression $[E]$, (iii) the type declaring a method $\mathit{Decl}(M)$, or (iv) the type declaring a field $\mathit{Decl}(F)$

of language constructs. These rules are essentially the same as in [20, 7], but rely on a predicate *isLibraryClass* to avoid the generation of constraints for: (i) calls to methods (constructors, static methods, and instance methods) declared in generic library classes, (ii) accesses to fields in generic library classes, and (iii) overriding relationships involving methods declared in generic library classes. Note the assumption that the program is already using a generic version of the library. Therefore, $[E]_{\mathcal{P}}$ may denote a generic type. Section 5 will discuss the generation of constraints that are counterparts to (i)–(iii) for references to generic library classes.

We now study a few of the constraint generation rules of Figure 5. Rule (1) states that an assignment $E_1 = E_2$ is type correct if the type of E_2 is the same as or a subtype of the type of E_1. For a field-access expression $E \equiv E_0.f$ that accesses a field F declared in class C, rule (2) defines the type of E to be the same as the declared type of F and rule (3) requires that the type of expression E_0 be a subtype of the type C in which F is declared. Here, the predicate $\mathit{IsLibraryClass}(C)$ is used to restrict the generation of these constraints to situations where class C is not a library type.

$$\frac{\mathcal{P} \text{ contains assignment } E_1 = E_2}{[E_2] \leq [E_1]} \tag{1}$$

$$\frac{\mathcal{P} \text{ contains field access } E \equiv E_0.f \text{ to field } F,\, C = \textit{Decl}(F),\, \neg\textit{IsLibraryClass}(C)}{\begin{array}{c}[E] = [F] \quad (2)\\ [E_0] \leq C \quad (3)\end{array}}$$

$$\frac{\mathcal{P} \text{ contains constructor call } E \equiv \texttt{new}\ C(E_1, \cdots, E_k)}{[E] = C} \tag{4}$$

$$\frac{\begin{array}{c}\mathcal{P} \text{ contains constructor call } \texttt{new}\ C(E_1, \cdots, E_k) \text{ to constructor } M,\\ \neg\textit{IsLibraryClass}(C),\, E'_i \equiv \textit{Param}(M, i),\, 1 \leq i \leq k\end{array}}{[E_i] \leq [E'_i]} \tag{5}$$

$$\frac{\begin{array}{c}\mathcal{P} \text{ contains call } E_0.m(E_1, \cdots, E_k) \text{ to virtual method } M,\\ \textit{RootDefs}(M) = \{\, C_1, \cdots, C_q \,\}\end{array}}{[E_0] \leq C_1 \textbf{ or} \cdots \textbf{or } [E_0] \leq C_q} \tag{6}$$

$$\frac{\begin{array}{c}\mathcal{P} \text{ contains call } E \equiv E_0.m(E_1, \cdots, E_k) \text{ to virtual method } M,\\ \neg\textit{IsLibraryClass}(\textit{Decl}(M)),\, E'_i \equiv \textit{Param}(M, i),\, 1 \leq i \leq k\end{array}}{\begin{array}{c}[E] = [M] \quad (7)\\ [E_i] \leq [E'_i] \quad (8)\end{array}}$$

$$\frac{\begin{array}{c}\mathcal{P} \text{ contains direct call } E \equiv C.m(E_1, \cdots, E_k) \text{ to static method } M,\\ \neg\textit{IsLibraryClass}(C),\, E'_i \equiv \textit{Param}(M, i),\, 1 \leq i \leq k\end{array}}{\begin{array}{c}[E] = [M] \quad (9)\\ [E_i] \leq [E'_i] \quad (10)\end{array}}$$

$$\frac{\mathcal{P} \text{ contains cast expression } E \equiv (C)E_0}{[E] = C} \tag{11}$$

$$\frac{\mathcal{P} \text{ contains down-cast expression } E \equiv (C)E_0,\, C \text{ is not an interface},\, [E_0]_{\mathcal{P}} \text{ is not an interface}}{C \leq [E_0]} \tag{12}$$

$$\frac{M \text{ contains an expression } E \equiv \texttt{this},\, C = \textit{Decl}(M)}{[E] = C} \tag{13}$$

$$\frac{M \text{ contains an expression } E \equiv \texttt{return}\ \ E_0}{[E_0] \leq [M]} \tag{14}$$

$$\frac{\begin{array}{c}M' \text{ overrides } M,\, 1 \leq i \leq \textit{NrParams}(M'),\, E_i \equiv \textit{Param}(M, i),\, E'_i \equiv \textit{Param}(M', i),\\ \neg\textit{IsLibraryClass}(\textit{Decl}(M))\end{array}}{\begin{array}{c}[E_i] = [E'_i] \quad (15)\\ [M'] \leq [M] \quad (16)\end{array}}$$

Fig. 5. Inference rules for deriving type constraints from various Java constructs.

Rules (6)–(8) are concerned with a virtual method call $E \equiv E_0.m(E_1, \cdots, E_k)$ that refers to a method M. Rule (6) states that a declaration of a method with the same signature as M must occur in some supertype of the type of E_0. The complexity in this rule stems from the fact that M may override one or more methods $M_1, \cdots, M_q$ declared in supertypes $C_1, \cdots, C_q$ of $\mathit{Decl}(M)$, and the type-correctness of the method call only requires that the type of receiver expression E_0 be a subtype of one of these C_i. This is expressed by way of a disjunction in rule (6) using auxiliary function *RootDefs* of Figure 3. Rule (7) defines the type of the entire call-expression E to be the same as M's return type. Further, the type of each actual parameter E_i must be the same as or a subtype of the type of the corresponding formal parameter E_i' (rule (8)).

Rules (11) and (12) are concerned with down-casts. The former defines the type of the entire cast expression to be the same as the target type C referred to in the cast. The latter requires this C to be a subtype of the expression E_0 being casted.

The constraints discussed so far are only concerned with type-correctness. Additional constraints are needed to ensure that program behavior is preserved. Rules (15) and (16) state that overriding relationships in $\mathcal{P}$ must be preserved in the refactored program (note that covariant return types are allowed in Java 1.5). Moreover, if a method $m(E_1, \cdots, E_k)$ overloads another method, then changing the declared type of any formal parameter E_i may affect the specificity ordering that is used for compile-time overload resolution [11]. To avoid such behavioral changes, we generate additional constraints $[E_i] = [E_i]_{\mathcal{P}}$ for all i $(1 \leq i \leq k)$ to ensure that the signatures of overloaded methods remain the same. Constraints that have the effect of preserving the existing type are also generated for actual parameters and return types used in calls to methods in classes for which source code cannot be modified.

5 Type Constraints for Generic Libraries

Additional categories of type constraints are needed for: (i) calls to methods in generic library classes, (ii) accesses to fields in generic library classes[6], and (iii) user classes that override methods in generic library classes. We first discuss a few concrete examples of these constraints, and then present rules that automate their generation.

5.1 New Forms of Type Constraints

Consider the call `m1.put(d1,list1)` on line (8) of Figure 1, which resolves to method `V Map<K, V>.put(K, V)`. This call is type-correct if: (i) the type of the first actual parameter, `d1`, is a subtype of the first actual type parameter of receiver `m1`, and (ii) the type of the second actual parameter, `list1`, is a subtype of the second actual type parameter of `m1`. These requirements are expressed by the constraints [`d1`]≤`K(m1)` and [`list1`]≤`V(m1)`, where the notation $T(E)$ is used for a new kind of constraint variable that denotes the value of actual type parameter T in the type of the expression E. Similar constraints are generated for return values of methods in generic library classes. For example, the call to `m2.get(d3)` on line (22) of Figure 1 refers to method `V Map<K, V>.get(Object)`. Here, the type of the entire expression has

[6] These can be handled in the same way as calls to methods in generic library classes, and will not be discussed in detail.

$$\frac{\mathcal{P} \text{ contains call } E_{rec}.\texttt{put}(E_{key}, E_{value}) \text{ to method } \texttt{V Map} < \texttt{K}, \texttt{V} > .\texttt{put}(\texttt{K}, \texttt{V})}{\begin{array}{r}[E_{key}] \leq \texttt{K}(E_{rec}) \\ [E_{value}] \leq \texttt{V}(E_{rec}) \\ [E_{rec}.\texttt{put}(E_{key}, E_{value})] = \texttt{V}(E_{rec})\end{array}} \quad [\texttt{put}]$$

$$\frac{\mathcal{P} \text{ contains call } E_{rec}.\texttt{get}(E_{key}) \text{ to method } \texttt{V Map} < \texttt{K}, \texttt{V} > .\texttt{get}(\texttt{Object})}{[E_{rec}.\texttt{get}(E_{key})] = \texttt{V}(E_{rec})} \quad [\texttt{get}]$$

$$\frac{\begin{array}{c}\mathcal{P} \text{ contains call } E_{rec}.\texttt{addAll}(E_{arg}) \text{ to method} \\ \texttt{boolean Collection} < \texttt{E} > .\texttt{addAll}(\texttt{Collection} <? \texttt{ extends E} >)\end{array}}{\begin{array}{l}[E_{arg}] \leq \texttt{Collection} \\ \texttt{E}(E_{arg}) \leq \texttt{E}(E_{rec})\end{array}} \quad [\texttt{addAll}]$$

Fig. 6. Constraint generation rules for calls to `V Map<K,V>.put(K,V)`, `V Map<K,V>.get(Object)`, and `boolean Collection<E>.addAll(Collection<? extends E>)`

the same type as the second actual type parameter of the receiver expression `m2`, which is expressed by: [`m2.get(d3)`] = `V`(`m2`). Wildcards are handled similarly. For example, the call `v1.addAll(list5)` on line (13) of Figure 1 resolves to method `boolean Collection<E>.addAll(Collection<? extends E>)`. This call is type-correct if the actual type parameter of `list5` is a subtype of the actual type parameter of receiver `v1`: `E`(`list5`) ≤`E`(`v1`).

Figure 6 shows rules that could be used to generate the type constraints for the calls to `put`, `get`, and `addAll` that were just discussed. Observe that the formal parameter of the `get` method has type `Object` and no relationship exists with the actual type parameter of the receiver expression on which `get` is called[7].

We generate similar constraints when a user class overrides a method in a generic library class, as was the case in the program of Figure 1 where `ListIterator.next()` overrides `Iterator.next()`. Specifically, if a user class C overrides a method in a library class L with a formal type parameter T, we introduce a new constraint variable $T(C)$ that represents the instantiation of L from which C inherits. Then, if a method M' in class C overrides a method M in L, and the signature of M refers to a type parameter T of L, we generate constraints that relate the corresponding parameter or return type of M' to $T(C)$.

For example, method `ListIterator.next()` on line (37) of Figure 1 overrides `Iterator<E>.next()`. Since the return type of `Iterator.next()` is type parameter `E`, we generate a constraint [`ListIterator.next()`] = `E(ListIterator)`. Note that this con-

[7] While it might seem more natural to define `get` as `V Map<K,V>.get(K)` instead of `V Map<K,V>.get(Object)`, this would require the actual parameter to be of type `K` at compile-time, and additional `instanceof`-tests and downcasts would need to be inserted if this were not the case. The designers of the Java 1.5 standard libraries apparently preferred the flexibility of being able to pass any kind of object over the additional checking provided by a tighter argument type. They adopted this approach consistently for all methods that do not write to a container (e.g., `contains`, `remove`, `indexOf`).

straint precisely captures the required overriding relationship because `ListIterator.next()` only overrides `Iterator<Integer>.next()` if the return type of `ListIterator.next()` is `Integer`.

5.2 Constraint Generation Rules for Generic Libraries

While type constraint generation rules such as those of Figure 6 can be written by the programmer, this is tedious and error-prone. Moreover, it is clear that their structure is regular, determined by occurrences of type parameters in signatures of methods in generic classes. Figure 7 shows rules for *generating constraints for calls to methods in generic classes*. For a given call, rule (r1) creates constraints that define the type of the method call expression, and rule (r2) creates constraints that require the type of actual parameters to be equal to or a subtype of the corresponding formal parameters. A recursive helper function *CGen* serves to generate the appropriate constraints, and is defined by case analysis on its second argument, $\mathcal{T}$. Case (c1) applies when $\mathcal{T}$ is a non-generic class, e.g., `String`. Case (c2) applies when $\mathcal{T}$ is a type parameter. In the remaining cases the function is defined recursively. Cases (c3) and (c4) apply when $\mathcal{T}$ is an upper or lower-bounded wildcard type, respectively. Finally, case (c5) applies when $\mathcal{T}$ is a generic type.

$$
\mathit{CGen}(\alpha, \mathcal{T}, E, \mathit{op}) = \begin{cases}
\{\alpha\ \mathit{op}\ C\} & \text{when } \mathcal{T} \equiv\ C \qquad \text{(c1)} \\
\{\alpha\ \mathit{op}\ T_i(E)\} & \text{when } \mathcal{T} \equiv\ T_i \qquad \text{(c2)} \\
\mathit{CGen}(\alpha, \tau, E, \leq) & \text{when } \mathcal{T} \equiv\ ?\ \mathit{extends}\ \tau \qquad \text{(c3)} \\
\mathit{CGen}(\alpha, \tau, E, \geq) & \text{when } \mathcal{T} \equiv\ ?\ \mathit{super}\ \tau \qquad \text{(c4)} \\
\{\alpha\ \mathit{op}\ C\} \cup & \text{when } \mathcal{T} \equiv C < \tau_1, \ldots, \tau_m > \qquad \text{(c5)} \\
\mathit{CGen}(W_i(\alpha), \tau_i, E, =) & \text{and } C \text{ is declared as} \\
 & C < W_1, \ldots, W_m >, 1 \leq i \leq m
\end{cases}
$$

$$
\frac{\mathcal{P} \text{ contains call } E \equiv\ E_{rec}.m(E_1, \ldots, E_k) \text{ to method } M, 1 \leq i \leq k}{\begin{array}{lr} \mathit{CGen}([E],\ [M]_{\mathcal{P}},\ E_{rec},\ =) \cup & \text{(r1)} \\ \mathit{CGen}([E_i],\ [\mathrm{Param}(M,i)]_{\mathcal{P}},\ E_{rec},\ \leq) & \text{(r2)} \end{array}}
$$

Fig. 7. Constraint generation rules for calls to methods in generic classes

We will now give a few examples that show how the rules of Figure 7 are used to generate type constraints such as those generated by the rules of Figure 6. As an example, consider again the call `m1.put(d1, list1)` to `V Map<K,V>.put(K,V)` on line 8 of the original program $\mathcal{P}$. Applying rule (r1) of Figure 7 yields *CGen*([`m1.put(d1, list1)`], `V`, `m1`, =), and applying case (c2) of the definition of *CGen* produces the set of constraints {[`m1.put(d1, list1)`] = `V(m1)`}. Likewise, for parameter `d1` in the call `m1.put(d1, list1)` on line 8, we obtain `m1.put(d1, list1)` $\overset{r2}{\Rightarrow}$ *CGen*([`d1`], K, `m1`, ≤) $\overset{c2}{\Rightarrow}$ {[`d1`] ≤ `K(m1)`}. Two slightly more interesting cases are the following:

line 13: $\texttt{v1.addAll(list5)} \overset{r2}{\Rightarrow}$
$\mathit{CGen}([\,\texttt{list5}\,], \texttt{Collection<? extends E>}, \texttt{v1}, \leq) \overset{c5}{\Rightarrow}$
$\{[\,\texttt{list5}\,] \leq \texttt{Collection}\} \cup \mathit{CGen}(\texttt{E}(\texttt{l}), \texttt{? extends E}, \texttt{v1}, =) \overset{c3}{\Rightarrow}$
$\{[\,\texttt{list5}\,] \leq \texttt{Collection}\} \cup \mathit{CGen}(\texttt{E}(\texttt{l}), \texttt{E}, \texttt{v1}, \leq) \overset{c2}{\Rightarrow}$
$\{[\,\texttt{list5}\,] \leq \texttt{Collection}\} \cup \{\texttt{E}(\texttt{l}) \leq \texttt{E}(\texttt{v1})\}$

line 19: $\texttt{m2.keySet()} \overset{r1}{\Rightarrow} \mathit{CGen}([\,\texttt{m2.keySet()}\,], \texttt{Set<K>}, \texttt{m2}, =) \overset{c5}{\Rightarrow}$
$\{[\,\texttt{m2.keySet()}\,] = \texttt{Set}\} \cup \mathit{CGen}(\texttt{E}(\texttt{m2.keySet()}), \texttt{K}, \texttt{m2}, =) \overset{c2}{\Rightarrow}$
$\{[\,\texttt{m2.keySet()}\,] = \texttt{Set}\} \cup \{\texttt{E}(\texttt{m2.keySet()}) = \texttt{K}(\texttt{m2})\}$

Table 1 below shows the full set of generics-related type constraints computed for the example program in Figure 1. Here, the appropriate rules and cases of Figure 7 are indicated in the last two columns.

Our algorithm also creates type constraints for methods in application classes that override methods in generic library classes. For example, the last row of Table 1 shows a type constraint required for the overriding of method `E Iterator<E>.next()` in class `ListIterator`. The rules for generating such constraints are similar to those in Figure 7 and have been omitted due to space limitations.

Table 1. Generics-related type constraints created for code from Figure 1. The labels in the two rightmost columns refer to rules and cases in the definitions of Figure 7

line	code	type constraint(s)	rule	cases
8	`m1.put(d1, list1)`	[`d1`] ≤ `K(m1)` [`list1`] ≤ `V(m1)` [`m1.put(d1, list1)`] = `V(m1)`	r2 r2 r1	c2 c2 c2
8	`m1.put(d2, list2)`	[`d2`] ≤ `K(m1)` [`list2`] ≤ `V(m1)` [`m1.put(d2, list2)`] = `V(m1)`	r2 r2 r1	c2 c2 c2
10	`v1.add(new Float(2.0))`	[`new Float(2.0)`] ≤ `E(v1)`	r2	c2
12	`list5.add(find(m1, 37))`	[`find(m1, 37)`] ≤ `E(list5)`	r2	c2
13	`v1.addAll(list5)`	[`list5`] ≤ `Collection` `E(list5)` ≤ `E(v1)`	r2	c5, c3, c2
15	`v2.add(v1)`	[`v1`] ≤ `E(v2)`	r2	c2
19	`m2.keySet()`	[`m2.keySet()`] = `Set` `E(m2.keySet())` = `K(m2)`	r1	c5, c2
19	`m2.keySet().iterator()`	[`m2.keySet().iterator()`] = `Iterator` `E(m2.keySet().iterator())` = `E(m2.keySet())`	r1	c5, c2
21	`it.next()`	[`it.next()`] = `E(it)`	r1	c2
22	`m2.get(d3)`	[`m2.get(d3)`] = `V(m2)`	r1	c2
37	override of `E Iterator<E>.next()`	[`ListIterator.next()`]= `E(ListIterator)`		

5.3 Closure Rules

Thus far, we introduced additional constraint variables such as $K(E)$ to represent the actual type parameter bound to K in E's type, and we described how calls to methods in generic libraries give rise to constraints on these variables. However, we have not yet discussed how types inferred for actual type parameters are constrained by language constructs such as assignments and parameter passing. For example, consider an assignment `a = b`, where `a` and `b` are both declared of type `Vector<E>`. The lack of covariance for Java generics implies that `E(a)` = `E(b)`. The situation becomes more complicated in the presence of inheritance relations between generic classes. Consider a situation involving class declarations[8] such as:

```
interface List<E_l> { ... }
class Vector<E_v> implements List<E_v> { ... }
```

and two variables, `c` of type `List` and `d` of type `Vector`, and an assignment `c = d`. This assignment can only be type-correct if the same type is used to instantiate $\mathtt{E}_l$ in the type of `c` and $\mathtt{E}_v$ in the type of `d`. In other words, we need a constraint $\mathtt{E}_l(\mathtt{c}) = \mathtt{E}_v(\mathtt{d})$. The situation becomes yet more complicated if generic library classes are assigned to variables of non-generic supertypes such as `Object`. Consider the program fragment:

```
Vector v1 = new Vector();
v1.add("abc");
Object o = v1;
Vector v2 = (Vector)o;
```

Here, we would like to infer $\mathtt{E}_v(\mathtt{v1}) = \mathtt{E}_v(\mathtt{v2}) = \mathtt{String}$, which would require tracking the flow of actual type parameters through variable `o`[9].

The required constraints are generated by a set of closure rules that is given in Figure 8. These rules infer, from an existing system of constraints, a set of additional constraints that unify the actual type parameters as outlined in the examples above. In the rules of Figure 8, α and α' denote constraint variables that are not type constants. Rule (17) states that, if a subtype constraint $\alpha \leq \alpha'$ exists, and another constraint implies that the type of α' or α has formal type parameter T_1, then the types of α and α' must have the same actual type parameter T_1[10]. This rule thus expresses the invariant subtyping among generic types. Observe that this has the effect of associating type parameters with variables of non-generic types, in order to ensure that the appropriate unification occurs in the presence of assignments to variables of non-generic types. For the example code fragment, a constraint variable $\mathtt{E}_v(\mathtt{o})$ is created by applying rule (17).

[8] In the Java collections library, the type formal parameters of both `Vector` and `List` have the same name, `E`. In this section, for disambiguation, we subscript them with v and l, respectively.

[9] In general, a cast to a parameterized type cannot be performed in a dynamically safe manner because type arguments are erased at run-time. In this case, however, our analysis is capable of determining that the resulting cast to `Vector<String>` would always succeed.

[10] Unless wildcard types are inferred, which we do not consider in this paper.

$$\frac{\alpha \leq \alpha' \qquad T_1(\alpha) \text{ or } T_1(\alpha') \text{ exists}}{T_1(\alpha) = T_1(\alpha')} \tag{17}$$

$$\frac{\begin{array}{c} T_1(\alpha) \text{ exists} \\ C_1\langle T_1 \rangle \texttt{ extends/implements } C_2\langle \mathcal{T} \rangle \\ C_2 \text{ is declared as } C_2\langle T_2 \rangle \end{array}}{\mathit{CGen}(T_2(\alpha), \mathcal{T}, \alpha, =)} \tag{18}$$

Fig. 8. Closure rules

Values computed for variables that denote type arguments of non-generic classes (such as `Object` in this example) are disregarded at the end of constraint solution.

Rule (18) is concerned with subtype relationships among generic library classes such as the one discussed above between classes `Vector` and `List`. The rule states that if a variable $T_1(\alpha)$ exists, then a set constraints is created to relate $T_1(\alpha)$ to the types of actual type parameters of its superclasses. Note that rule (18) uses the function *CGen*, defined in Figure 7. For example, if we have two variables, `c` of type `List` and `d` of type `Vector`, and an initial system of constraints $[\,\texttt{d}\,] \leq [\,\texttt{c}\,]$, and $\texttt{String} \leq \texttt{E}_v\texttt{(d)}$, then using the rules of Figure 8, we obtain the additional constraints $\texttt{E}_v(\texttt{d}) = \texttt{E}_v(\texttt{c})$, $\texttt{E}_l(\texttt{d}) = \texttt{E}_v(\texttt{d})$, $\texttt{E}_l(\texttt{c}) = \texttt{E}_l(\texttt{d})$ and $\texttt{E}_l(\texttt{c}) = \texttt{E}_v(\texttt{d})$.

We conclude this section with a remark about special treatment of the `clone()` method. Although methods that override `Object.clone()` may contain arbitrary code, we assume that implementations of `clone()` are well-behaved (in the sense that the returned object preserves the type arguments of the receiver expression) and generate constraints accordingly.

6 Constraint Solving

Constraint solution involves computing a set of legal types for each constraint variable and proceeds in standard iterative fashion. In the initialization phase, an initial type estimate is associated with each constraint variable, which is one of the following: (i) a singleton set containing a specific type (for constants, type literals, constructor calls, and references to declarations in library code), (ii) the singleton set $\{ B \}$ (for each constraint variable $K(E)$ declared in library code, where K is a formal type parameter with bound B, to indicate that E should be left raw), or (iii) the type universe (in all other cases). In the iterative phase, a work-list is maintained of constraint variables whose estimate has recently changed. In each iteration, a constraint variable α is selected from the work-list, and all type constraints that refer to α are examined. For each type constraint $t = \alpha \leq \alpha'$, the estimates associated with α and α' are updated by removing any element that would violate t, and α and/or α' are reentered on the work-list if appropriate (other forms of type constraints are processed similarly). As estimates monotonically decrease in size as constraint solution progresses, termination is guaranteed. The result of this process is a set of legal types for each constraint variable.

Since the constraint system is typically underconstrained, there is usually more than one legal type associated with each constraint variable. In the final solution, there needs to be a singleton type estimate for each constraint variable, but the estimates for different constraint variables are generally not independent. Therefore, a single type is chosen from each non-singleton estimate, after which the inferencer is run to propagate that choice to all related constraint variables, until quiescence. The optimization criterion of this step is nominally to select a type that maximizes the number of casts removed. As a simple approximation to this criterion, our algorithm selects an arbitrary most specific type from the current estimate (which is not necessarily unique). Although overly restrictive in general (a less specific type may suffice to remove the maximum number of casts/warnings), and potentially sub-optimal, the approach appears to be quite effective in practice. The type selection step also employs a filter that avoids selecting "tagging" interfaces such as `java.lang.Serializable` that define no methods, unless such are the only available choices[11].

In some cases, the actual type parameter inferred by our algorithm is equal to the bound of the corresponding formal type parameter (typically, `Object`). Since this does not provide any benefits over the existing situation (no additional casts can be removed), our algorithm leaves raw any declarations and allocation sites for which this result is inferred. The opposite situation, where the actual type parameter of an expression is completely unconstrained, may also happen, in particular for incomplete programs. In principle, any type can be used to instantiate the actual type parameter, but since each choice is arbitrary, our algorithm leaves such types raw as well.

There are several cases where raw types must be retained to ensure that program behavior is preserved. When an application passes an object *o* of a generic library class to an external library[12], nothing prevents that library from writing values into *o*'s fields (either directly, or by calling methods on *o*). In such cases, we cannot be sure what actual type parameter should be inferred for *o*, and therefore generate an additional constraint that equates the actual type parameter of *o* to be the bound of the corresponding formal type parameter, which has the effect of leaving *o*'s type raw. Finally, Java 1.5 does not allow arrays of generic types [4] (e.g., type `Vector<String>[]` is not allowed). In order to prevent the inference of arrays of generic types, our algorithm generates additional constraints that equate the actual type parameter to the bound of the corresponding formal type parameter, which has the effect of preserving rawness.

Constraint solution yields a unique type for each constraint variable. Allocation sites and declarations that refer to generic library classes are rewritten if at least one of its inferred actual type parameters is more specific than the bound of the corresponding formal type parameter. Other declarations are rewritten if their inferred type is more specific than its originally declared type. Any cast in the resulting program whose operand type is a subtype of its target type is removed.

[11] Donovan et al. [8] apply the same kind of filtering.

[12] The situation where an application receives an object of a generic library type from an external library is analogous.

7 Implementation

We implemented our algorithm in the context of Eclipse, using existing refactoring infrastructure [3], which provides abstract syntax trees (with symbol binding resolution), source rewriting, and standard user-interface componentry. The implementation also builds on the type constraint infrastructure that was developed as part of our earlier work on type-related refactorings [20]. Much engineering effort went into making the refactoring scalable, and we only mention a few of the most crucial optimizations. First, a custom-built type hierarchy representation that allows subtype tests to be performed in constant time turned out to be essential. This is currently accomplished by maintaining, for each type, hash-based sets representing its supertypes and subtypes. However, we plan to investigate the use of more space-efficient mechanisms [22]. Second, as solution progresses, certain constraint variables are identified as being identically constrained (either by explicit equality constraints, or by virtue of the fact that Java's generic types are invariant, as was discussed in Section 2). When this happens, the constraint variables are *unified* into an equivalence class, for which a single estimate is kept. A union-find data structure is used to record the unifications in effect as solution progresses. Third, a compact and efficient representation of type sets turned out to be crucially important. Type sets are represented using the following expressions (in the following, S denotes a set of types, and t, t' denote types): (i) *universe*, representing the universe of all types, (ii) $\mathit{subTypes}(S)$, representing the set of subtypes of types in S, (iii) $\mathit{superTypes}(S)$, representing the set of supertypes of types in S, (iv) $\mathit{intersect}(S, S)$, (v) $\mathit{arrayOf}(S)$, representing the set of array types whose elements are in S, and (vi) $\{t, t', \ldots\}$, i.e., explicitly enumerated sets. In practice, most subtype queries that arise during constraint solving can be reduced to expressions for which obvious closed forms exist, and relatively few sets are ever expanded into explicitly represented sets. Basic algebraic simplifications are performed as sets are created, to reduce their complexity, as in $\mathit{intersect}(\mathit{subTypes}(S), S)) = \mathit{subTypes}(S)$. Fourth, we use a new Eclipse compiler API that has been added to improve performance of global refactorings, by avoiding the repeated resolution of often-used symbol bindings.

The refactoring currently supports three modes of operation. In *basic* mode arbitrary declarations may be rewritten, and precise parametric supertypes may be inferred for user-defined subtypes of generic library classes. In *noderived* mode, arbitrary declarations may be rewritten, but we do not change the supertype of user-defined subtypes of generic library classes. The *preserve erasure* mode is the most restrictive because it does not change the supertype of user-defined subtypes of generic library classes and it preserves the erasure of all methods. In other words, it only adds type arguments to declarations and hence preserves binary compatibility.

The forthcoming Eclipse 3.1 release will contain a refactoring called INFER GENERIC TYPE ARGUMENTS, which is largely based on the concepts and models presented in this paper and has adopted important parts of the research implementation. Currently, only the *preserve erasure* mode is supported.

8 Experimental Results

We evaluated our method on a suite of moderate-sized Java programs[13] by inferring actual type parameters for declarations and allocation sites that refer to the standard collections. In each case, the transformed source was validated using Sun's `javac` 1.5 compiler. Table 2 states, for each benchmark, the number of types, methods, total source lines, non-blank non-comment source lines, and the total number of declarations, allocation sites, and casts. We also give the number of allocation sites of generic types, generic-typed declarations, subtypes of generic types, and "unchecked warnings."

We experimented with the three modes—*basic*, *noderived*, and *preserve erasure*—that were discussed in Section 7. The results of running our refactoring on the benchmarks appear in Table 3. The first six columns of the figure show, for each of the three

Table 2. Benchmark characteristics

benchmark	benchmark size							generics-related measures			
	types	methods	LOC	NBNC LOC	decls	allocs	casts	allocs	decls	subtypes	warnings
JUnit	59	382	5,265	2,394	1,012	305	54	24	48	0	27
V_poker	35	279	6,351	3,097	1,044	198	40	12	27	1	47
JLex	22	121	7,842	4,333	668	146	71	17	33	1	40
DB	32	222	8,594	3,363	939	225	78	14	36	1	652
JavaCup	36	302	11,087	3,833	1,065	341	595	19	62	0	55
TelnetD	52	397	11,239	3,219	995	128	46	16	28	0	22
Jess	184	756	18,199	7,629	2,608	654	156	47	64	1	692
JBidWatcher	264	1,830	38,571	21,226	5,818	1,698	383	76	184	1	195
ANTLR	207	2,089	47,685	28,599	6,175	1,163	443	46	106	3	84
PMD	395	2,048	38,222	18,093	5,163	1,066	774	75	286	1	183
HTMLParser	232	1,957	50,799	20,332	4,895	1,668	793	72	136	2	205
Jax	272	2,222	53,897	22,197	7,266	1,280	821	119	261	3	158
xtc	1,556	5,564	90,565	37,792	14,672	3,994	1,114	330	668	1	583

Table 3. Experimental results

benchmark	casts removed			unchecked warnings remaining			program entities rewritten			*(basic)*	time (sec.)
	basic	*noderived*	*preserve erasure*	*basic*	*noderived*	*preserve erasure*	generic allocs	generic decls	all decls	generic subtypes	*(basic)*
JUnit	24	24	21	2	2	8	24	57	79	0	9.9
V_poker	32	25	25	0	0	1	12	31	31	1	8.4
JLex	48	47	47	6	6	6	16	28	29	1	5.7
DB	40	40	37	0	634	634	13	32	43	1	8.7
JavaCup	488	488	486	2	2	2	19	70	81	0	9.0
TelnetD	38	38	37	0	0	0	15	27	30	0	6.8
Jess	83	83	82	9	642	642	42	58	68	1	15.9
JBidWatcher	207	204	177	5	5	25	74	195	238	3	64.5
ANTLR	86	84	82	5	7	8	45	80	202	1	32.1
PMD	154	135	132	21	35	36	64	278	322	9	42.0
HTMLParser	172	170	168	7	13	13	70	154	220	2	34.6
Jax	158	139	132	82	82	82	87	188	301	2	45.4
xtc	398	394	327	71	73	136	315	664	1,138	3	113.9

[13] For more details, see: `www.junit.org`, `www.cs.princeton.edu/~appel/modern/java/JLex/`, `www.cs.princeton.edu/~appel/modern/java/CUP/`, `www.spec.org/osg/jvm98/`, `vpoker.sourceforge.znet`, `telnetd.sourceforge.net`, `www.antlr.org`, `jbidwatcher.sourceforge.net`, `pmd.sourceforge.net`, `htmlparser.sourceforge.net` and `www.ovmj.org/xtc/`.

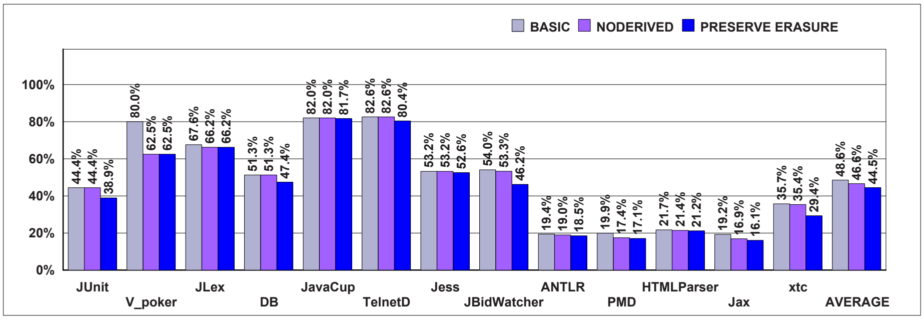


Fig. 9. Percentages of casts removed, for each of the three modes

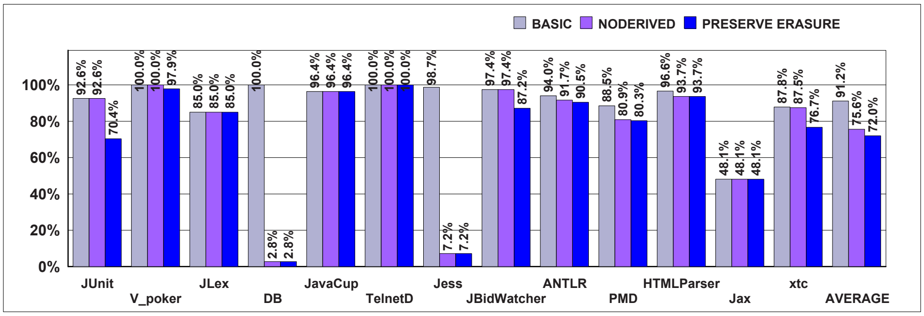


Fig. 10. Percentages of unchecked warnings eliminated, for each of the three modes

modes, the number of casts removed and unchecked warnings eliminated. The next four columns show, for the *basic* mode only, the number of generic allocation sites rewritten, the number of generic declarations rewritten, the total number of declarations rewritten, and the number of user-defined subtypes for which a precise generic supertype is inferred, respectively. The final column of the figure shows the total processing time in *basic* mode (the processing times for the other modes are similar). Processing *xtc*, our largest benchmark, took slightly under two minutes on a 1.6GHz Pentium M[14] and about 500Mb of heap space. These results clearly demonstrate our algorithm's scalability and we expect our technique to scale to programs of 500 KLOC or more.

8.1 Casts Removed

Figure 9 shows a bar chart that visualizes the percentage of casts removed in each benchmark, for each of the three modes. As can be seen from this figure, the *basic* mode removes an average of 48.6% of all casts from each benchmark, the *noderived* mode is slightly less effective with an average of 46.6% of all casts removed, and the

[14] The processing time for *xtc* can be broken down as follows: 26.6 seconds for constraint generation, 71.1 seconds for constraint solving, and 16.3 seconds for source rewriting.

preserve erasure mode is the least effective with 44.5% of all casts removed. When considering these numbers, the reader should note that the total number of casts given in Table 2 includes casts that are not related to the use of generic types. However, a manual inspection revealed that our tool removes the vast majority of generics-related casts, from roughly 75% to 100%. For example, we estimate that only one-fifth of *ANTLR*'s total number of casts relates to the use of collections, which is close to our tool's 19.4% removal rate.

8.2 Unchecked Warnings Eliminated

A clearer indication of the effectiveness of our algorithm is apparent in the high proportion of "unchecked warnings" eliminated. This statistic is a rough measure of the improvement in the degree of type safety in the subject program. Figure 10 visualizes the percentage of unchecked warnings eliminated in each benchmark, for each of the three modes. As can be seen from this figure, the *basic* mode eliminates an average of 91.2% of all unchecked warnings for each benchmark, followed by the *noderived* mode with an average of 75.6% and the *preserve erasure* mode with 72.0%. Note that the lower averages for the *noderived* and and *preserve erasure* mode are largely due to the very low percentages of unchecked warnings removed on the *DB* and *Jess* benchmarks. We will discuss these cases in detail shortly.

8.3 Analysis of Results

We conducted a detailed manual inspection of the source code of the refactored benchmarks, in order to understand the limitations of our analysis. Below is a list of several issues that influenced the effectiveness of our analysis.

Arrays. Several benchmarks create arrays of collections. For example, *JLex* creates an array of `Vectors`, and *xtc* creates several arrays of `HashMaps`. Since Java 1.5 does not permit arrays of generic types, raw types have to be used, resulting in several unchecked warnings, and preventing some casts from being removed (8 casts in the case of *JLex*).

Wildcard Usage. Several benchmarks (*JBidWatcher*, *HTMLParser*, *JUnit*, *Jax* and *xtc*) override library methods such as `java.lang.ClassLoader.loadClass()` that return wildcard types such as `java.lang.Class<?>`. Our method is incapable of inferring wildcard types, and leaves the return types in the overriding method definitions raw, resulting in unchecked warnings.

Polymorphic Containers. In several benchmarks (*JBidWatcher*, *Jax*, *Jess*, and *xtc*), unrelated types of objects are stored into a container. In such cases, the common upper bound of the stored objects is `java.lang.Object`, and the reference is left raw. The most egregious case occurs in *Jax*, where many different `Hashtables` are stored in a single local variable. Splitting this local variable prior to the refactoring results in the elimination of an additional 71 unchecked warnings.

Use of clone(). Various benchmarks (*JBidWatcher*, *JUnit*, *JavaCup*, *Jess*, *ANTLR*, and *xtc*) invoke the `clone()` method on container objects, and cast the result to a raw container type. Although our analysis tracks the flow of types through calls to `clone()`, rewriting the cast is not helpful, because the compiler would still produce a warning[15]. Our tool does not introduce casts to parameterized types, which means that unchecked warnings will remain.

Static Fields. The *xtc* benchmark contains 11 references to `Collections.EMPTY_LIST`, a static field of the raw type `List`. Several declarations will need to remain raw, resulting in unchecked warnings. It is interesting to note that the Java 1.5 standard libraries provide a generic method `<T> List<T> emptyList()` that enables polymorphic use of a shared empty list.

User-Defined Subtypes of Generic Library Classes. In most cases, the inference of precise generic supertypes for user-defined subclasses of generic library classes has little impact on the number of casts removed and warnings eliminated. However, the *DB* and *Jess* benchmarks both declare a subclass `TableOfExistingFiles` of `java.util.Hashtable` that contains 600+ calls of the form `super.put(s1,s2)`, where `s1` and `s2` are `Strings`. In *basic* mode, `TableOfExistingFiles` is made a subclass of `Hashtable<String,String>` and the unchecked warnings for these super-calls are eliminated. In the *noderived* and *preserve erasure* modes, `TableOfExistingFiles` remains a subclass of raw `Hashtable`, and a warning remains for each call to `put`, thus explaining the huge difference in the number of unchecked warnings.

9 Context Sensitivity

Conceptually, our analysis can be extended with context-sensitivity by simply generating multiple sets of constraints for a method, one for each context. In principle, this can result in tighter bounds on parametric types when collections are used in polymorphic methods, and in the removal of more casts. Moreover, we could introduce type parameters on such polymorphic methods to accommodate their use with collections with different type parameters.

Figure 11(a) shows an example program that illustrates this scenario using a method `reverse()` for reversing the contents of a `Vector`. The `reverse()` method is invoked by methods `floatUse()` and `intUse()`, which pass it `Vectors` of `Floats` and `Integers`, respectively. Applying the previously presented analysis would determine that both vectors reach method `reverse()` and infer an element type that is a common upper bound of `Float` and `Integer` such as `Number`. Therefore, all allocation sites and declarations in the program would be rewritten to `Vector<Number>`, and neither of the two casts could be removed.

However, if we create two analysis contexts for `reverse`—one for each call site—then one can infer bounds of `Float` and `Integer` for the two creation sites of vec-

[15] While casts to parameterized types such as `Vector<String>` are allowed in Java 1.5, such casts will succeed if the expression being casted is an instance of the corresponding erased type (`Vector`), and compilers produce a warning to inform users of this unintuitive behavior.

```
class ContextExample {
  void floatUse() {
    Vector v =
      new Vector();
    v.add(new Float(3.14));
    reverse(v);
    Float f = (Float)v.get(0);
  }
  void intUse() {
    Vector v =
      new Vector();
    v.add(new Integer(6));
    reverse(v);
    Integer i = (Integer)v.get(0);
  }
  void reverse(Vector v) {
    for(int i=0;i<v.size()/2;i++){
      Object temp = v.get(i);
      v.set(i,v.get(v.size() - i));
      v.set(v.size() - i,temp);
    }
  }
}
```

(a)

```
class ContextExample {
  void floatUse() {
    Vector<Float> v =
      new Vector<Float>();
    v.add(new Float(3.14));
    reverse(v);
    Float f = v.get(0);
  }
  void intUse() {
    Vector<Integer> v =
      new Vector<Integer>();
    v.add(new Integer(6));
    reverse(v);
    Integer i = v.get(0);
  }
  <T> void reverse(Vector<T> v) {
    for(int i=0;i<v.size()/2;i++){
      T temp = v.get(i);
      v.set(i,v.get(v.size() - i));
      v.set(v.size() - i,temp);
    }
  }
}
```

(b)

Fig. 11. Example program that illustrates the need for context-sensitive analysis

tors. Conceptually, this is equivalent to analyzing a transformed version of the program that contains two clones of the `reverse()` method, one of which is called from `intUse()`, the other from `floatUse()`. The two contexts of `reverse` would receive different type estimates for parameter `v`, and our code transformation could exploit this information by transforming `reverse()` into a generic method, and remove both casts. This result is shown in Figure 11(b).

We implemented a context-sensitive version of the previously presented algorithm, in which we used a low-cost variant of Agesen's Cartesian Product Algorithm [1, 2] to determine when different contexts should be created for a method, and reported the results in a previous technical report [19]. To our surprise, we could not find any non-synthetic benchmarks where the use of context-sensitive analysis resulted in the removal of additional casts. We believe that there are two major reasons why context-sensitive analysis was not useful. The first is that the standard libraries already provide a rich set of functionality, and there is relatively little need for writing additional helper methods. Second, the relatively few applications that do define helper methods that operate on collections tend to use these methods monomorphically. An investigation of larger applications might turn up more opportunities for context-sensitive analysis, but it is our conjecture that there will be relatively few such opportunities.

10 Related Work

The work most closely related to ours is that by Donovan et al. [8], who also designed and implemented a refactoring for migrating an application to a generic version of a class library that it uses. Like us, Donovan et al. evaluate their algorithm by inferring generic types for occurrences of the standard collections in a set of Java benchmarks, and measure their success in terms of the number of casts that can be removed. There are a number of significant differences between the two approaches.

First, the approach by Donovan et al. relies on a context-sensitive pointer analysis[16] based on [24, 1] to determine the types stored in each allocation site for a generic library class. Moreover, Donovan et al. create "guarded" constraints that may or may not be applied to the type constraint system depending on the rawness of a particular declaration, and their solving algorithm may require (limited) backtracking if such a rawness decision leads to a contradiction later on. Our approach is simpler because it requires neither context-sensitive analysis nor backtracking, and therefore has greater potential for scaling to large applications. The differences in observed running times seem to bear this out (Donovan et al. report a running time of 462 seconds on the ∼27 KLOC *HTMLParser* using a 3GHz Pentium 4 with 200Mb heap, while our tool requires 113.9 seconds on a ∼90 KLOC program using a 1.6GHz Pentium M using 512Mb heap).

Second, there are several differences in the kinds of source transformations allowed in the two works: (i) Donovan et al. restrict themselves to transformations that do not affect the erasure of a class, while our approach allows the modification of declarations, (ii) Donovan's work was done prior to the release of Java 1.5 and their refactoring tool conforms to an earlier specification of Java generics, which does not contain wildcard types and which allows arrays of generic types, and (iii) our method is capable of inferring precise generic supertypes for subtypes of generic library classes that are defined in application code (see, e.g., Figure 2 in which we infer that class `MyIterator` extends `Iterator<String>`). Third, our tool is more practical because it is fully integrated in a popular integrated development environment.

For a more concrete comparison, we manually inspected the source generated by both tools for 5 of the 7 benchmarks analyzed in [8]: *JLex*, *JavaCup*, *JUnit*, *V_poker* and *TelnetD*. A head-to-head comparison on *ANTLR* and *HTMLParser* was impossible due to differences in the experimental approach taken[17].

In most cases, our tool was able to remove the same or a higher number of generics-related casts than did Donovan's, in a small fraction of the time. The differences in casts removed derive from several distinct causes. First, our tool's ability to infer type parameters for user-defined subtypes of parametric types permits the removal of additional casts (e.g., 6 additional casts could be removed in *V_poker* in clients of a local class extending `Hashtable`). Second, Donovan's tool was implemented before the final Java 1.5 specification was available and conforms to an early draft, in which parameterized types were permitted to be stored in arrays; the final specification, however, requires that such generic types be left raw. As a result, Donovan's tool infers non-raw types for certain containers in *JLex* that our tool (correctly) leaves raw, preventing the

[16] The context-sensitive variant of our algorithm [19] discussed in Section 9 is also based on the Cartesian Product Algorithm, but it uses context-sensitivity for a different purpose, namely to identify when it is useful to create generic methods.

[17] Donovan et al. identified classes in these benchmarks that could be made generic, manually rewrote them accordingly, and treated them as part of the libraries. As a result, the number of removed casts that they report cannot be directly compared to ours, as it includes casts rendered redundant by the generics that they manually introduced.

removal of certain casts. Third, our algorithm models `Object.clone()` so that type parameter information is not lost across the call boundary. As a result, our tool removes all 24 generics-related casts from *JUnit*, while Donovan's tool only removes 16.

Von Dincklage and Diwan [23] address the problems of converting non-generic Java classes to use generics (parameterization) and updating non-generic usages of generic classes (instantiation). Their approach, like ours, is based on constraints. Von Dincklage's tool employs a suite of heuristics that resulted in the successful parameterization of several classes from the Java standard collections. However, the code of those classes had to be manually modified to eliminate unhandled language constructs before the tool could be applied. The tool's correctness is based on several unsound assumptions (e.g., public fields are assumed not to be accessed from outside their class, and the type of the argument to `equals` is assumed to be identical to the receiver's type), and it can alter program behavior by modifying virtual method dispatch due to changed overriding relationships between methods. No results are given about how successful the tool is in instantiating non-generic classes with generic information.

The problem of introducing generic types into a program to broaden its use has been approached before by several researchers. Siff and Reps [18] focused on translating C functions into C++ function templates by using type inference to detect latent polymorphism. In this work, opportunities for introducing polymorphism stem from operator overloading, references to constants that can be wrapped by constructor calls, and from structure subtyping. Duggan [9] gives an algorithm (not implemented) for genericizing classes in a small Java-like language into a particular polymorphic variant of that language. This language predated the Java 1.5 generics standard by several years and differs in a nontrivial number of respects. Duggan does not address the problem of migrating non-generic code to use generics. The programming environments CodeGuide [6] and IntelliJ IDEA [12] provide "Generify'' refactorings that are similar in spirit to ours. We are not aware of the details of these implementations, nor of the quality of their results.

11 Conclusions and Future Work

We have presented a refactoring that assists programmers with the adoption of a generic version of an existing class library. The method infers actual type parameters for declarations and allocation sites that refer to generic library classes using an existing framework of type constraints. We implemented this refactoring in Eclipse, and evaluated the work by migrating a number of moderate-sized Java applications that use the Java collections framework to Java 1.5's generic collection classes. We found that, on average, 48.6% of the casts related to the use of collections can be removed, and that 91.2% of the unchecked warnings are eliminated. Our approach distinguishes itself from the state-of-the-art [8] by being more scalable and by its ability to accommodate user-defined subtypes of generic library classes. The "Infer Generic Type Arguments" in the forthcoming Eclipse 3.1 release is largely based on the concepts presented in this paper, and has adopted important parts of our implementation.

Plans for future work include the inference of wildcard types. As indicated in Section 8.3, doing so will help remove additional casts and unchecked warnings.

Acknowledgments

We are grateful to Alan Donovan and Michael Ernst for many useful discussions about comparisons between our two algorithms, and for sharing information about the benchmarks used in [8]. The anonymous ECOOP reviewers, Michael Ernst, and Bartek Klin provided many helpful comments and suggestions.

References

1. AGESEN, O. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proc. of ECOOP* (1995), pp. 2–26.
2. AGESEN, O. *Concrete Type Inference: Delivering Object-Oriented Applications*. PhD thesis, Stanford University, December 1995.
3. BÄUMER, D., GAMMA, E., AND KIEŻUN, A. Integrating refactoring support into a Java development tool. In *OOPSLA'01 Companion* (October 2001).
4. BRACHA, G., COHEN, N., KEMPER, C., ODERSKY, M., STOUTAMIRE, D., THORUP, K., AND WADLER, P. Adding generics to the Java programming language, final release. Tech. rep., Java Community Process JSR-000014, September 2004.
5. BRACHA, G., ODERSKY, M., STOUTAMIRE, D., AND WADLER, P. Making the future safe for the past: Adding genericity to the Java programming language. In *Proc. of OOPSLA* (1998), pp. 183–200.
6. Omnicore codeguide. `http://www.omnicore.com/codeguide.htm`.
7. DE SUTTER, B., TIP, F., AND DOLBY, J. Customization of Java library classes using type constraints and profile information. In *Proc. of ECOOP* (2004), pp. 585–610.
8. DONOVAN, A., KIEŻUN, A., TSCHANTZ, M., AND ERNST, M. Converting Java programs to use generic libraries. In *Proc. of OOPSLA* (Vancouver, BC, Canada, 2004), pp. 15–34.
9. DUGGAN, D. Modular type-based reverse engineering of parameterized types in Java code. In *Proc. of OOPSLA* (1999), pp. 97–113.
10. FOWLER, M. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999.
11. GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. *The Java Language Specification (3rd Edition)*. Addison-Wesley, 2000.
12. JetBrains IntelliJ IDEA. `http://www.intellij.com/idea/`.
13. IGARASHI, A., PIERCE, B. C., AND WADLER, P. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS 23*, 3 (2001), 396–450.
14. LANGER, A., AND KREFT, K. Arrays in Java Generics. Manuscript `http://www.langer.camelot.de`.
15. MUNSIL, W. Case study: Converting to Java 1.5 type-safe collections. *Journal of Object Technology 3*, 8 (2004), 7–14.
16. ODERSKY, M., AND WADLER, P. Pizza into Java: Translating theory into practice. In *Proc. of POPL* (1997), pp. 146–159.
17. PALSBERG, J., AND SCHWARTZBACH, M. *Object-Oriented Type Systems*. John Wiley & Sons, 1993.
18. SIFF, M., AND REPS, T. W. Program generalization for software reuse: From C to C++. In *Foundations of Software Engineering* (1996), pp. 135–146.
19. TIP, F., FUHRER, R., DOLBY, J., AND KIEŻUN, A. Refactoring techniques for migrating applications to generic Java container classes. Tech. Rep. Research Report RC 23238, IBM Research, June 2004.
20. TIP, F., KIEŻUN, A., AND BÄUMER, D. Refactoring for generalization using type constraints. In *Proc. of OOPSLA* (Anaheim, CA, 2003), pp. 13–26.

21. Torgersen, M., Hansen, C. P., Ernst, E., von der Ahé, P., Bracha, G., and Gafter, N. M. Adding wildcards to the Java programming language. In *Proc. of ACM Symposium on Applied Computing (SAC)* (Nicosia, Cyprus, 2004), pp. 1289–1296.
22. Vitek, J., Horspool, R. N., and Krall, A. Efficient type inclusion tests. In *Proc. of OOPSLA* (1997), pp. 142–157. *SIGPLAN Notices* 32(10).
23. von Dincklage, D., and Diwan, A. Converting Java classes to use generics. In *Proc. of OOPSLA* (Vancouver, BC, Canada, 2004), pp. 1–14.
24. Wang, T., and Smith, S. Precise constraint-based type inference for Java. In *Proc. of ECOOP* (2001), pp. 99–117.

Sharing the Runtime Representation of Classes Across Class Loaders

Laurent Daynès and Grzegorz Czajkowski

Sun Microsystems Laboratories
{Laurent.Daynes, Grzegorz.Czajkowski}@sun.com

Abstract. One of the most distinctive features of the Java^{TM} programming language is the ability to specify class loading policies. Despite the popularity of class loaders, little has been done to reduce the cost associated with defining the same class by multiple loaders. In particular, implementations of the Java virtual machine (JVM^{TM}) create a complete runtime representation of each class regardless of how many class loaders already define the same class. This lack of sharing leads to poor memory utilization and to replicated run-time work. Recent efforts achieve some degree of sharing only when dynamic binding behaves predictably across loaders. This limits sharing to class loaders whose behavior is fully controlled by the JVM. As a result applications that implement their own class loading policies cannot enjoy the benefit of sharing.

We present a novel technique for sharing the runtime representation of classes (including bytecodes and, under some conditions, compiled code) across arbitrary user-defined class loaders. We describe how our approach is applied to the multi-tasking virtual machine (MVM). The new multi-tasking virtual machine retains the fast start-up time of the original MVM while extending the scope of footprint savings to applications that exploit user-defined class loaders.

1 Introduction

One of the most distinctive features of the Java^{TM} programming language [10] is the ability to define class loading policies [13]. Class loaders are exploited in a wide range of applications, such as scripting environments, IDEs, runtime code injection tools, aspect-oriented programming platforms, web browsers, servlet engines, and application servers.

Class loaders are popular for several reasons. They provide separate namespaces, which allows a program to link components regardless of whether they include different versions of the same classes or different classes with the same name. This feature enables the implementation of a form of isolation, where carefully crafted software modules can be loaded multiple times without interfering with one another [4]. Class loaders also give programs control over the location where classes are loaded from, and an opportunity to transparently enhance third-party code by providing mechanisms for its interception and modification (via bytecode transformations) before it is linked with the rest of the managed execution environment.

Using class loaders comes at a cost, however. JVM implementations typically replicate the entire runtime representation of a class in memory for each class loader that defines the class[1]. Defining a class multiple times also replicates the effort to create an optimized runtime representation, by repeating class file parsing, construction of main-memory data structures, bytecode verification, (optional) bytecodes quickening, resolution of constants, and identification and recompilation of frequently used methods.

The use of delegation relationships between class loaders, where one class loader can delegate the definition of a class to another, helps to limit these problems but can rapidly become error prone as the complexity of the delegation relationships increases. Besides, not all uses of class loaders can be accommodated with delegation. For example, delegation is inadequate when multiple instances of the same software component must be loaded and isolated from one another.

The cost of user-defined class loaders is a consequence of the inability of current JVM implementations to share the main-memory representation of a class between multiple definitions of that class. JVMs typically construct the executable image of a program at runtime in several incremental steps: first by loading class files from locations specified by their loaders and building corresponding main memory representations, then by linking them to other classes as symbolic references are encountered during method execution, and eventually by compiling performance-critical methods. This evolution of a program image is a consequence of dynamic binding and of the use of an architecture-neutral format of class files, as required by the specification of the Java programming language [10] (JLS). Both make the building of a sharable image at runtime difficult and prevent applying well-established shared library techniques used for less dynamic programming languages (e.g. [3]).

Several recent efforts have achieved some degree of sharing of the runtime representation of classes between executing programs [5, 6, 19, 8]. However, they can only do so when dynamic binding has a predictable behavior across loaders, e.g., when a symbolic link from a class A to a class B is guaranteed to resolve identically across all class loaders. Thus, sharing is only supported for those class loaders whose behavior is fully controlled by the JVM. Sharing is not supported when classes are defined by user-defined loaders. Systems based on static compilation (e.g., [20]) face similar issues (see Section 6).

This paper presents a novel technique for sharing the runtime representation of classes between multiple arbitrary defining class loaders, and describes its application to the Multi-Tasking Virtual Machine (or MVM) [5]. This new implementation of MVM is called hereafter CLSVM (*Class Loader Sharing Virtual Machine*). CLSVM is a significant step forward in the technology of transpar-

[1] The character strings representing symbols are usually shared across representations of all classes though.

ent sharing of safe language meta-data, and improves on MVM by bringing the benefits of sharing to user-defined class loaders.

CLSVM's sharing of the runtime representation of a class is orthogonal to sharing by delegation. Sharing by delegation makes a class type visible to multiple loaders and has an impact on program semantics, while sharing of the runtime representation of classes is transparently and automatically performed by the JVM, has no impact on program behavior, and does not violate or impact type safety.

In CLSVM, sharing is achieved by splitting the runtime representation of a class into loader-dependent and loader-independent parts, and by making bytecode interpretation *loader re-entrant* so that the bytecodes of methods are sharable across multiple loaders. The loader-dependent part is replicated for each loader that defines the class. Classes loaded by the bootstrap loader are treated specially in order to exploit the predictability of symbolic link resolution and consequently to maximize sharing.

A consequence of such a design is that some of the class loading and run-time compilation effort is shared across multiple class loaders. In particular, class file parsing, and most of the building of a main-memory runtime representation of the class is done only once. Post-processing of bytecodes that may take place at link time is also done once.

CLSVM's dynamic compiler mixes two strategies to reduce the overhead of dynamic compilations. Whenever possible, the compiler attempts to share compiled code across multiple defining loaders. When sharing is not possible, the compiler constructs a new version of the compiled code. Even in this situation opportunities for compilation time savings arise: only if the method has not been compiled at all is it compiled from its bytecodes. Otherwise, instead of recompiling the method, the compiler clones the compiled code and modifies its loader-dependent part.

The rest of this paper is organized as follows. Section 2 describes principles for sharing between multiple loaders. Section 3 details a prototype implementation of CLSVM based on MVM, which in turn extends the Java HotSpot^{TM} virtual machine [16] (referred to as HSVM). Section 4 discusses how dynamic compilations take advantage of sharing across loaders. Section 5 reports a quantitative assessment of the impact of sharing on the performance and memory footprint of programs written in the Java programming language (or *Java programs*). Related work is discussed in Section 6. Section 7 summarizes the contributions of this work.

2 Design Overview

Sharing an element of the runtime representation of a class between class loaders is possible when the element is independent of its defining loaders. Loader dependencies arise from symbolic links to other classes, which may resolve at runtime to different class definitions from one loader to another, and from references from a class's runtime representation to data that are private to a loader, such as instances of `java.lang.Class` or static variables. At first sight, a significant amount of the runtime representation of a class appears independent of the loader that defines it. In particular, it should be possible to share across two loaders, each defining a class described by the same class file, the bytecodes of that class, the constant part of its constant pool (i.e., the part that can never resolve to different objects at runtime, such as constant values), and meta-data describing the class itself (e.g., tables holding description of fields, methods, and exceptions).

However, several obstacles make sharing across multiple loaders difficult. First, loaders may resolve the super class of the same class differently. This may result in different object layouts (since one super-class may declare more fields than the other, and of different types), and different virtual tables (since one super-class may declare a different number of methods, with different signatures, and different methods may be overridden). Second, interpreters often exploit resolved symbolic links to rewrite bytecodes into faster versions that need not test whether links should be resolved or whether classes should be initialized. Such *quickened* bytecodes become, in effect, loader-dependent. The dynamic compiler also exploits information derived from resolved symbolic links and makes compiled code loader dependent. These obstacles taken together seem to contradict the first intuition that a significant amount of meta-data can be shared across loaders.

Our design overcomes these problems as follows. First, sharing is allowed only if some conditions on inheritance are met, so as to avoid dealing with cases where object layout and virtual tables would be different. Second, the interpretation of bytecodes is made loader re-entrant by re-organizing the runtime representation of classes so as to efficiently access each loader's private data, and by adding *barriers* to guarantee that link resolutions and class initializations are performed upon their first use by a loader. Third, the dynamic compiler maintains information to help determine if native code can be shared between loaders, and if not, to help build a new version of compiled code without paying the cost of a full-blown compilation from the method's bytecodes.

2.1 Terminology and Notation

We use the terminology and notation of [13] to describe relations between classes and class loaders. Namely, a class type is denoted as $< C, L_d >^{L_i}$, where C denotes the name of the class, L_d denotes the class's *defining* loader, and L_i denotes its *initiating* loader. The initiating loader of a class (i.e., the loader invoked to load the class) is not necessarily the loader that defines the class, due to the API of class loaders that enables one loader to delegate the definition of a class to another. The simplified notation $< C, L_d >$ is used when the initiating loader of a class is not relevant. Similarly, C^{L_i} denotes a situation when the defining loader is not relevant. By definition $< C, L_1 >=< C, L_2 >\Rightarrow L_1 = L_2$.

We extend this notation with the $\sim$ operator to denote that two class types satisfy the *same sharing conditions* (see Section 2.2) and can therefore share their runtime representation. By definition $< C, L_1 >\sim< C, L_2 >\Rightarrow< C, L_1 >\neq< C, L_2 >$ (this restriction simplifies discussion that follows). For convenience, we introduce the following notation: $< C, L_1 > \cong < C, L_2 >$ to denote $< C, L_1 > \sim < C, L_2 > \vee < C, L_1 > = < C, L_2 >$.

2.2 Sharing Conditions

Let us consider two distinct loaders L_1 and L_2, each defining a class C. The conditions that enable $< C, L_1 > \sim < C, L_2 >$, that is, the sharing of the runtime representations of $< C, L_1 >$ and $< C, L_2 >$, are defined below.

The first condition for $< C, L_1 > \sim < C, L_2 >$ is that the class files submitted to the JVM by L_1 and L_2 must be identical. For simplicity, two class files (either residing on a disk or dynamically generated) are considered identical if the classes are byte-per-byte equal, although this is a fairly coarse method for determining if two class files encode the same class definition. A more precise implementation would require parsing the class files and determining equivalence of the declarations. This approach is substantially more expensive and, in practice, is unlikely to be much more effective.

The second condition requires that the super-classes S^{L_1}, S^{L_2} of $< C, L_1 >$ and $< C, L_2 >$ respectively are such that $S^{L_1} \cong S^{L_2}$.

The third condition is that $< C, L_1 >$ and $< C, L_2 >$ have the same abstract methods. Note that this property holds if the two sharing conditions above are satisfied and if C does not implement any interfaces. The third condition is specifically directed at classes that implement interfaces. It guarantees that all classes that share their runtime representation have the same *unimplemented methods*, that is, methods that are not defined by the class but yet are declared in an interface. For example, let us consider a class C that implements an interface I that declares a single method m. Method m is unimplemented if neither C nor any of its super-classes implement m. In this case, m must be treated as a public abstract method of C.

The sharing conditions guarantee several properties that simplify sharing. First, they guarantee that $< C, L_1 >$ and $< C, L_2 >$ have the same number of static variables, that their respective instances have the same number of fields (whether directly defined by the class or inherited), and that fields with the same name have the same signature and will be at the same offset. This property allows the JVM to lay out identically the instances of classes whose runtime representation is shared, to share reference maps used for garbage collections, and to share the descriptions of fields. Second, the sharing conditions guarantee that the methods of $< C, L_1 >$ and $< C, L_2 >$, whether declared directly by these classes or inherited, have the same name, signature, protection level, and bytecodes, and that they can be assigned the same index in a virtual method table (the Java programming language inheritance model enables table-driven implementation of virtual method dispatch).

Although the sharing conditions guarantee that $< C, L_1 >$ and $< C, L_2 >$ implement the same number of interfaces, and have the same unimplemented methods, they do not mandate that these interfaces declare exactly the same methods. Consider the example in Figure 1. In this case, the sharing conditions allow the runtime representation of C to be shared across L_1 and L_2, although they implement different interfaces.

```
interface A { // In L1            interface A { // In L2
   int foo(int i);                  Integer foo(Integer i);
   long foo(long l);                void bar(int i);
   void bar(int i);
}                                 }
abstract class C implements A { // Defined by both L1 and L2
   int foo(int i){...}
   Integer foo(Integer i){...}
   long foo(long l){...}
} // bar is the only unimplemented method in both L1 and L2
```

Fig. 1. The sharing conditions can be satisfied despite different interface definitions

3 The Design of CLSVM

CLSVM extends MVM, the multi-user, multi-tasking virtual machine [5, 7] with the ability to share meta-data across user-defined class loaders. The parts of the design of MVM relevant to CLSVM are briefly reviewed before discussing the actual design of CLSVM.

3.1 Background on MVM

MVM is an implementation of the JVM that co-locates the execution of multiple programs in a single operating system process. Each program execution is carried out by an *isolate* [11]. Isolates provide a program with the illusion of a standalone JVM: programs have the same behavior as if they were running on a JVM of their own. Each isolate has its own primordial loader and hierarchy of class loaders. No sharing of objects can take place between isolates and the JVM safeguards against most types of inter-isolate interference.

MVM substantially reduces the footprint of programs by implementing a form of sharing that we call *task re-entrance.* Task re-entrance is supported only for classes defined by class loaders whose behavior is fully controlled by MVM, that is, the *primordial* and *system* loader of each isolate.

The primordial loader is a special class loader that bootstraps the class loading mechanism. It is used to load the *base* classes that are intimately associated with a JVM implementation and are essential to its functioning (such as classes of the `java.*` packages). The system loader is the loader that defines the main class of a program. It typically obtains class files from the local file system at a fixed location specified at program start-up. MVM forces this location to be the same for all programs it executes, and requires that class files stored there remain unchanged for the duration of its execution. The system loader serves class loading requests by first delegating them to the primordial loader, and only defines classes that the primordial loader failed to define. This behavior is predictible, and a class loaded by a primordial or a system loader of any task is always built from the same class file. Further, symbolic references from classes defined by a primordial or a system loader always resolve identically across all tasks.

This allows for a simplified form of sharing where only the mutable state part of the runtime representation of a class (e.g., static variables, class initialization state, protection domain, instance of `java.lang.Class` etc.) needs to be replicated per loader. In particular, information derived from resolved symbolic links, such as field offsets, virtual table indexes, static method addresses, etc., can be shared across loaders, further increasing the amount of sharing. In MVM no form of sharing is supported for classes defined by program-defined loaders.

Like MVM, CLSVM implements task re-entrance for classes loaded by primordial loaders. For all user-defined loaders CLSVM implements *loader re-entrance*, which allows sharing of the runtime representation of a class with any other class that satisfies the sharing conditions described earlier. The type of re-entrance implemented for system loaders can be chosen at start-up time: by default, CLSVM uses task re-entrance, like in MVM. All other aspects of MVM, such as isolate management and termination, per-isolate garbage collection, and fast inter-isolate communication, are left unchanged in CLSVM.

The rest of this section focuses on the following aspect of loader re-entrance: (i) how to organize the runtime representation of a class so as to maximize sharing while minimizing its overhead, (ii) how to efficiently retrieve loader-private information from shared code, and (iii) how to make the interpretation of shared bytecodes loader re-entrant.

3.2 Runtime Representation of Classes

The runtime representation of a class consists of data structures that mirror the architecture-neutral binary representation of that class, in a main memory format optimized for the various sub-systems of the JVM.

In CLSVM, the runtime representation of a class is split in a loader-independent and a loader-dependent representation (*LIR* and *LDR*, respectively). Loaders that satisfy the sharing conditions for a class share the same LIR, but each has its own LDR for the class. LIRs include a reference to a LDR *template*. The template serves two functions: it is a blueprint for constructing an LDR, and, to minimize space overhead, it is always used as the LDR of one loader.

Figure 2 illustrates this organization. It depicts the runtime representation of two classes satisfying the same sharing conditions: $< B, L_1 >$, whose LDR acts as a template, and $< B, L_2 >$ that was built using $< B, L_1 >$.

The LIR contains most of the runtime representation of a class. It consists of a `sharedRep` object which includes a reference map for garbage collection, references to an array of fields declared by the class template, to a shared constant pool, and to the LDR currently used as a template.

Each `sharedRep` of a LIR S also includes a reference to the *super* `sharedRep` of the LIR shared by the runtime representation of the super-classes of all classes that have S for their LIR. Recall that the second sharing condition requires that the super-classes of two classes $< C, L_1 >$ and $< C, L_2 >$ such that $< C, L_1 > \sim < C, L_2 >$ are either the same class or share their runtime representation. In either case, this means that the `sharedReps` of the super-classes of $< C, L_1 >$ and $< C, L_2 >$ are the same.

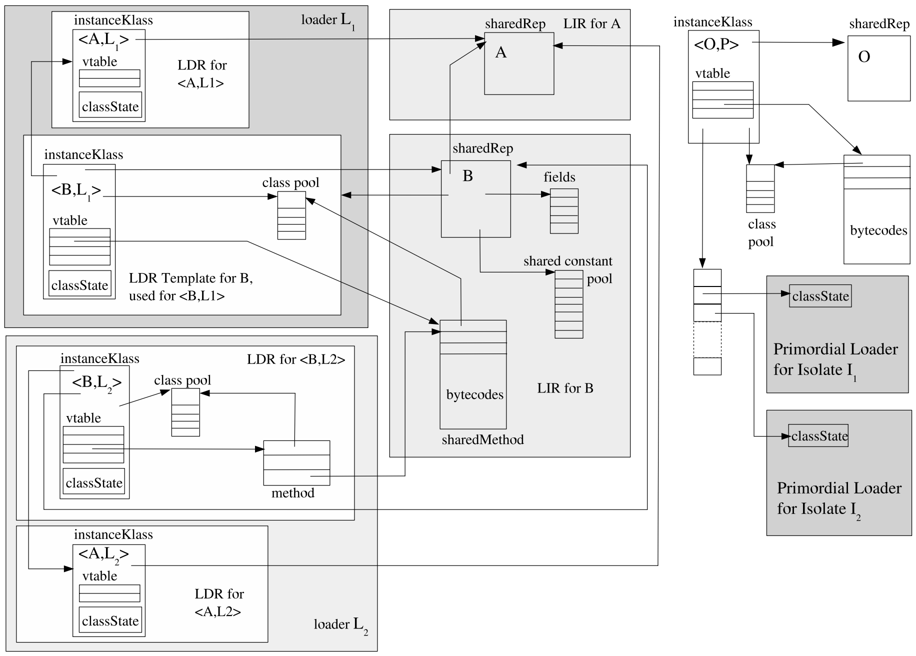


Fig. 2. CLSVM runtime representation of classes

The LDR of a class consists of an `instanceKlass` object, which includes storage for its virtual method table (vtable) and its interface table (itable), and a reference to the `sharedRep` object it was built from. The runtime representation of classes that are task re-entrant is structured slightly differently in that most of the LDR is also shared across the task re-entrant loaders, except for non-re-entrant state, such as the class's static variables, the instance of `java.lang.Class` and `java.lang.ClassLoader` for the class, and its initialization state. This non-re-entrant state is embedded in the `instanceKlass` object for loader re-entrant classes. This arrangement is not possible for task re-entrant classes since the whole `instanceKlass` object is shared among all isolates. Instead, the non-re-entrant state of each isolate sharing the `instanceKlass` is stored in a separate object accessible via a table indexed by isolate identifiers; the reference to the table is stored in the `instanceKlass`.

An example of this is shown on Figure 2: a class O is defined by the primordial loaders of two isolates, I_1 and I_2. The resulting class type $< O, P >$ is represented by the same `instanceKlass` for both I_1 and I_2, and only task-dependent state is replicated for each task. Regardless of whether their class is task re-entrant or not, class instances (i.e., Java objects) contain in their header a pointer to the `instanceKlass` that represents their class.

As mentioned earlier, the sharing conditions guarantee that the vtable indexes are the same across all loaders that satisfy them. They also guarantee that methods are inherited and overridden in exactly the same way across class loaders. However, entries in vtables must refer to loader-dependent method representations. Methods have two runtime representations: a loader-dependent representation, called a method object, and a shared method object. Method objects consist of an invocation counter and references to a shared method object, to a class pool, and to native code produced by the dynamic compiler, if any (see Section 4). Shared method objects encapsulate the bulk of a method definition, most notably, the method bytecodes. All of the shared method object is loader-independent, except for a loader-specific header. This header comprises the same information as in method objects, except for the shared method object reference. This organization allows one `instanceKlass` (typically, the LDR template's) to use the shared methods directly (like$< B, L_1 >$ on Figure 2), thus avoiding the space overhead of systematically splitting the runtime representation of a method.

The interpreter and the compiler dynamically resolve links using the class pool, the shared constant pool, and the constant pool cache of the class that defines the method being executed. The class pool and the shared constant pool are built directly from the constant pool defined in a class file. The class pool is filled only with symbolic links to classes. All other constant pool entries are entered into the shared constant pool, which contains only loader-independent information, namely indexes to class pool or shared constant pool entries, constant numerical and string values, or symbols. Loader-dependent information, other than symbolic links to classes, is confined to the constant pool cache, constructed at class link time. It contains information built from resolved symbolic links to fields, methods, and interfaces to enable faster interpretation of some bytecodes.

3.3 Class Loading

Classes loaded by the JVM are recorded in a *system dictionary* that maps keys composed of a fully qualified class name and a class loader reference to an `instanceKlass`. Multiple entries in the dictionary can refer to the same `instanceKlass` as a result of delegation between loaders.

A *shared class repository* maps linked lists of `sharedRep` objects to unique fingerprints computed over the bytes of class files. All `sharedRep`s in a given linked list are constructed from class files that all have the same fingerprint value. Having more than one shared runtime representation of a class for the same class file can occur because of possible violations of the sharing conditions between defining loaders (e.g., if the shared representation of the super classes of the defined classes are different). Fingerprints are computed as SHA-1 digests [18] of class files.

The class loading machinery uses the system dictionary and the shared class repository to determine whether a loader's request for defining a class can use an existing shared runtime representation for the newly defined class. When instructed by a loader to define a class, CLSVM fetches a class file from the specified input stream and computes its SHA-1 digest. The digest is used to retrieve all the shared representations of classes that were built with a class file of equal value. Note that the format of the class file does not need verification before computing the SHA-1 digest, since if the specified class file does not conform to a valid class file format, its digest cannot map to an existing entry of the shared class repository. If the digest does not map to any `sharedRep`, the format of the class file is verified and parsed to create a new `sharedRep` object, which is then entered into the repository.

If a linked list of `sharedRep` objects is found in the repository, an element that satisfies the sharing conditions for the defining loader is looked up. Let L be that loader, and $< C, L >$ the class type being defined. The first of the sharing conditions already holds since all `sharedRep`s from the list have a digest equal to that of the class file submitted by the loader. To test the second condition, the system dictionary is searched for the `instanceKlass` of $< C, L >$'s super class. If the search is not successful, class loading proceeds recursively to load the super-class. Once the `instanceKlass` of the super class is retrieved, its reference to its `sharedRep` is compared with the `super` reference of the `sharedRep` under evaluation. The second sharing condition holds if the two references are equal.

Lastly, $< C, L >$ must define the same unimplemented methods as other classes already sharing the evaluated `sharedRep`. Because the first two sharing conditions already hold, only unimplemented methods of $< C, L >$'s declared interfaces need to be verified. Hence the third condition is automatically satisfied if $< C, L >$ does not declare any interfaces. If it does, then unimplemented methods are looked up in the `sharedRep` and compared to those of $< C, L >$. The third condition holds if the number of unimplemented methods and their names and types are the same.

If none of the existing `sharedRep`s satisfies the sharing condition for $< C, L >$, a new one must be created. Creating a new `sharedRep` in this case does not require parsing the class file. Instead, an existing `sharedRep` is cloned and changed only in those places that depend on the super class and the unimplemented abstract methods, since a violation of the sharing conditions corresponds to having different values for some of these.

3.4 Sharing Bytecodes

The runtime representation of classes described above is not sufficient to allow sharing of methods. Bytecode interpretation must also be made loader reentrant. This requires efficient access to a class loader's copy of static variables, and proper triggering of link resolution and class initialization once for each loader that shares the bytecodes. Both are achieved by using the constant pool cache associated with the private representation of the *current class*, i.e., the class that defines the method being executed. The reference to the current class is stored, upon method invocation, in a dedicated location on the invoked method's stack frame. Short sequences of instructions called barriers trigger link resolution and class initialization.

A link resolution barrier (LRB) is required for all bytecodes that refer to a loader-dependent symbolic link. In CLSVM, the bytecodes in this category are the quickened versions of `getfield`, `putfield`, `invokevirtual`, `invokespecial`, `invokeinterface`. LRBs are redundant in presence of class initialization barriers, so the former is not necessary if the latter is already required. A class initialization barrier (CIB) is needed when interpreting the quickened versions of the four bytecode instructions that might trigger class initialization: `getstatic`, `putstatic`, `invokestatic` and `new`.

Both LRBs and CIBs work along the same principle: the operand of a quickened bytecode is an index to an entry of the current class's constant pool cache that holds information necessary to interpret the bytecode (e.g., a field offset, a vtable index, etc.). The information is initialized with a distinguishable marker that is tested by the interpreter. On SPARC® processors, these barrier tests add only a single branch on register value. For instance, offsets and vtable index information are typically initialized to a negative value so that LRBs just consist of a single branch on negative value, as shown below:

```
ld        [Rcache + // Retrieve offset to field
            (header_size + 2*wordSize)], Roffset
brgz,pt   Roffset, resolved // LRB
ld        [ Robject + Roffset], Rvalue // load field
```

Upon detecting the marker by a barrier, execution is routed to a stub that calls the runtime to perform the action associated with the barrier (link resolution or class initialization). Before resuming interpreted execution, the marker is replaced with the information needed by the interpreter in the constant pool entry, so that subsequent interpretation of bytecodes indexing that entry with the same class loader will not trigger link resolution. The first interpretation of the same bytecode instruction, but on behalf of a different loader, will trigger the barrier again since each loader uses a distinct constant pool cache.

4 Impact on Dynamic Compilation

HSVM mixes bytecode interpretation with dynamic compilation to achieve high-performance. Methods are initially interpreted. Per-method invocation counters are incremented on each invocation to detect frequently called methods. Methods that reach a given threshold of interpreted invocations are compiled. Subsequent invocations result in executing their compiled code.

This approach can be improved so that the effort of compiling the methods of a class is amortized across the loaders that define it. One possible strategy is to make the code produced by the dynamic compiler loader re-entrant, so that different defining loaders of the same method *always* share the same native code. This strategy has several advantages. First, compilation costs are paid only once per shared representation of a method, no matter how many loaders define that method, hence the cost of compilation is amortized across loaders. Second, because the code is re-entrant, it can be used immediately by any loader defining the method, eliminating bytecode interpretation. Third, memory footprint is reduced by sharing the compiled code across loaders.

However, making compiled code loader re-entrant introduces some overhead otherwise eliminated by a dynamic compiler. Dynamic compilation exploits the runtime knowledge of resolved links to remove the overhead of dynamic linking. For example, a dynamic compiler can determine the offset of a field of an object and generate a simple load instruction that does not use any meta-information (such as the runtime constant pool cache) at runtime. Such optimizations are not possible with loader re-entrant code because a level of indirection is required wherever a symbolic link to another class is used. So, for instance, loading a field of an object requires determining, at runtime, the current loader and then finding out the offset to the field in the context of that loader. Whereas the impact of such indirection may be benign to the performance of interpreted methods, it may be prohibitive in compiled code.

CLSVM employs a three-pronged strategy for dynamic compilation that mixes task re-entrance, cloning, and sharing of loader-dependent code.

As in MVM, methods of task re-entrant classes are compiled into task re-entrant code. Such code is produced by adding class initialization barriers before every possible first use of a class, and generating code to access static variables in a task re-entrant way. That is, static variables are retrieved from the class's table of per-isolate class state using an isolate identifier stored in a current thread's descriptor (see [5] for details). Past experience with MVM showed that the impact of task re-entrance on performance is negligible compared with the benefits of sharing compiled code for method of classes defined by task re-entrant loaders.

For methods of loader re-entrant classes, the compiler produces code optimized for a particular set of loader dependencies, i.e., using information derived from symbolic links as resolved by a particular loader. The code is annotated along the way with information identifying the loader-dependent sequences of instructions and what they depend on. The annotations are then used to determine if the code can be used as is by other loaders, or if a new version should be produced according to a new set of dependencies. In the latter case, instead of compiling the method for each defining loader from scratch, the compiler clones an existing version of it, and modifies its loader-dependent part only. This makes generation of native code faster as steps for parsing bytecodes, building an intermediate representation, performing optimization, and generating code are avoided.

Loader-dependent code sharing is based on the observation that the loader-dependent information derived from symbolic links may be constant across loaders that share the runtime representation of the class that defines the compiled method. For example, the offset to an instance variable of a class B is constant across all defining loaders of B that satisfy the same sharing conditions. In other words, a symbolic link to an instance variable of B from a class A is constant across two loaders L_1 and L_2 that share the runtime representation of A (i.e., $< A, L_1 > \sim < A, L_2 >$) if $B^{L_1} \cong B^{L_2}$. Thus, if the only symbolic links used in a method m of A are to instance variables of B, the compiled code for m can be shared between $< A, L_1 >$ and $< A, L_2 >$. In this strategy a method is compiled from bytecodes only once, no matter how many loaders define its class. Once compiled, native code for the method is obtained through cloning or sharing.

The compiler assumes the linkages of the loader that triggered the compilation, and exploits information obtained from resolved links, effectively making the native code dependent on a (potentially empty) set of resolved links. To enable sharing and cloning, the location of each loader-dependent sequence of native instructions is recorded along with the position of the corresponding bytecode instruction. This position offers a compact and loader-independent way of documenting a dependency: given a bytecode position and a loader-dependent method representation, one can retrieve the type of its dependency (e.g., offset of an instance field), and the class the dependency refers to (through the class pool of the method's class). Thus, code annotations consist of pairs of offsets: an offset to the first instruction of a loader-dependent sequence of native code, and another offset to a bytecode instruction. These pairs are recorded in a table of dependencies.

The table of dependencies is used to determine if native code listed in the shared method, and produced for a particular loader, can be used by another loader that shares the runtime representation of the method. Sharing of native code is possible if there are no dependencies, or if loaders have exactly the same dependencies. For example, a method that only manipulates instance variables of its class is loader-independent since the sharing conditions guarantee that offsets to these fields are the same across loaders that satisfy the same sharing conditions. In another example, a method may refer only to symbols of classes defined by the primordial loader in all loaders. In this case, the native code for the method can be shared between loaders since symbolic links to methods and variables of task re-entrant classes will refer to exactly the same item in all loaders.

The sharing of compiled code can be permitted even if the classes referred to by the code are not the same. For example, let us consider classes A and B:

```
class A {
  private int x;
  private static int X;
  int getx(){ return x;}
  int getxX(){ return x * X;}
  int foo(C c){ return x * c.z;}
  int bar(){ return x * B.Y;}
}
```

```
class B {
  static int Y = 94;
}
```

Let us further assume that class A has been defined by two loaders L_1 and L_2 such that $< A, L_1 > \sim < A, L_2 >$. Although $< A, L_1 > \neq < A, L_2 >$, the native code produced for method `getx` can be shared between $< A, L_1 >$ and $< A, L_2 >$ since the only dependency of the compiled code is the offset to the instance variable `x`, which is guaranteed by the sharing conditions (see Section 2.2) to be the same for both loaders. Such symbolic link references do not need to be recorded in the dependency table. By contrast, the native code produced for the `getxX` method cannot be shared between the loaders as it depends on the address of the static variable `X` which differs for each defining loader. The case for `foo` is more subtle: if $C^{L_1} \cong C^{L_2}$, then the native code can be shared since

Table 1. Modifications required for each type of dependencies to adapt a clone of a compiled method to a new loader L_r is the requesting loader, L_u is an owner of the original compiled code

Type of symbolic link	Conditions for leaving code unchanged	What to change if condition is false
instance variable	$C^{L_r} \cong C^{L_u}$	offset in load/store instruction
dynamically bound method	$C^{L_r} \cong C^{L_u}$	reset inline cache for virtual method/interface invocation
static variable	$C^{L_r} = C^{L_u}$	address of static variable
class	$C^{L_r} = C^{L_u}$	class address and instance size in immediate value register load
statically bound method	$C^{L_r} = C^{L_u}$	address of method entry point in call instruction

`z` resolves to the same offset for both loaders, either because $C^{L_1} = C^{L_2}$, or because the sharing conditions guarantee this. Otherwise, the method cannot be shared. Similarly, the native code for the `bar` method can be shared between L_1 and L_2 if $B^{L_1} = B^{L_2}$. More generally, let L_r be a loader that requests native code for a method m of class A, L_u the loader of one of the users of native code of method m, and C a class on which m depends. Table 1 lists, for each dependency type, the conditions for leaving the corresponding instructions unchanged, and what changes are required otherwise. If no condition is violated, no change is needed, and the code can be shared by L_u and L_r.

To determine whether any changes to the code are required, the compiler iterates over the dependent class listed in the dependency table. For each dependent class C, the compiler first determines if the link to C^{L_r} has been resolved, by examining the entry in L_r's class pool at the index recorded in the dependency table. If the link to C^{L_r} is not resolved, the determination for code sharing cannot be made, and sharing is prohibited. The compiler then selects the condition that determines if code change is needed based on the dependencies for class C (see also Table 1). If the condition $C^{L_r} = C^{L_u}$ is required, the compiler compares the references to C^{L_r} and C^{L_u} obtained from the class pool of $< A, L_r >$ and $< A, L_u >$ respectively, at the index recorded in the dependency table. If the condition $C^{L_r} \sim C^{L_u}$ is required, the compiler compares the references to the `sharedRep` objects from C^{L_r} and C^{L_u}.

To clone the native code of a method, the compiler first copies the native code and walks over the dependency table to identify classes for which changes are required in the native code. For each such class, the compiler iterates over the corresponding dependencies section. Using the bytecode position recorded there it retrieves the corresponding bytecode from the method template. Each such bytecode maps to a function that implements the logic for modifying the sequence of native code according to the operands of the bytecode instruction and the class pool of the recipient of the clone.

Since native code cloning or sharing is cheaper than compilation from bytecodes, switching from bytecode interpretation to native code execution takes place earlier for methods that have been already compiled.

5 Experiments

This section compares MVM and CLSVM with respect to memory footprint, program start-up time, and application execution time.

The aim of the presented techniques is to reduce the footprint of programs that extensively rely on user-defined class loaders, by sharing the runtime representation of classes across loaders of the same or of different programs. Another equally important goal is to avoid performance and start-up time regression with respect to MVM. Our prototype of CLSVM implements task re-entrance for both the primordial and system loaders, and loader re-entrance for user-defined loaders.

CLSVM is a derivative of MVM, which in turn derives from HSVM (the Java HotSpot^{TM} virtual machine [16]) version 1.3.1 with the client compiler. All results are reported relative to HSVM. The experiments were performed on a Sun Blade 1000^{TM} equipped with 8 GB of main memory and two UltraSPARC® III+ processors clocked at 1015 MHz, running the Solaris^{TM} 10 Operating Environment.

5.1 Start-Up Time

In both MVM and CLSVM programs can be started either from the command line or directly (programmatically) by an isolate. The former option involves the creation of a process that communicates with a *login* isolate to request the launching of a new isolate and to establish input/output bindings between the isolate and the initiating process (see [7] for further details).

Start-up time can be approximated by running an empty program (i.e., one whose `main()` methods consists of just a `return` statement). Table 2 reports the results for CLSVM and MVM, for both ways of starting a program (*cli* for command line start-up, and *java* for start-up from within an isolate). The results are expressed as speed-up with respect to the time to execute the empty program with HSVM (e.g., in *cli*, MVM starts a program 1.84 times faster than HSVM). They indicate that support for loader re-entrance has no negative impact on start-up performance. This is expected since none of the features of loader re-entrance are exercized at start-up.

Table 2. Start-up improvements relative to HSVM

	cli	java
MVM	1.84	26.82
CLSVM	1.85	26.21

5.2 Footprint

To quantify the impact of the design of CLSVM on memory usage, we experimented with two popular real-world applications: Apache Ant (version 1.6.2), and Apache Tomcat servlet engine (version 4.2.1). The latter was used to run JSPWiki [2], a Wiki clone implemented with Java Server Pages [1]. Tomcat maintains a hierarchy of class loaders to allow its components and applications it hosts to access different repositories of available classes and resources. Ant uses class loaders in a similar fashion, to customize per-user access to different libraries and resources.

Another common use of class loaders is to transparently inject code at runtime, either for profiling purposes or to enhance application code with an aspect (e.g., persistence). Typically a loader uses a bytecode editing library to transform the contents of fetched class files and submits the modified bytecodes to define the class. Our third experiment, referred to as the *bytecode transformation workload*, emulates this behavior using Apache's BCEL toolkit. We applied it to programs from the SPECjvm98 suite. The transformation was relatively simple: counting the number of dynamic (run-time) accesses to static variables by application code.

Memory footprint measurements were obtained with the help of the `pmap` command from the Solaris Operating Environment and then correlated with the JVM-specific runtime information regarding its use of virtual memory regions. Memory accounting was thus accurate for all memory regions, such as the heap area used for application data, runtime representations of classes, and compiled code area. The numbers reported exclude memory regions shared across processes, such as read-only parts of shared libraries and read-only memory-mapped jar files.

The data in the following figures were obtained as follows. For non-server tests (Ant, SPECjvm98 benchmarks), multiple instances of a given program were executed in sequence. Each program instance was artificially kept alive through a shutdown hook. Each hook would send a notification to an external supervising process then wait for an answer before exiting. Upon receiving a shutdown hook notification, the supervising process would obtain memory usage before starting the next program instance. Commands to exit were sent to the shutdown hooks only once all programs had been executed and their memory usage captured.

Recourse to shutdown hooks to maintain Wiki servers alive is unnecessary since these stay up and running until explicitly instructed to terminate. Therefore, the experiment proceeded by starting one Wiki server, submitting 100 requests, and capturing memory usage before repeating this sequence up to the desired number of servers. The same mix of 100 requests was sent to each server. It consisted of requests for pages' content, for editing them, and either saving or cancelling the edits.

Figure 3 shows the data obtained for Ant (up to 5 instances) and Tomcat/JSPWiki (up to 5 servers). The left-hand part of the figure shows memory

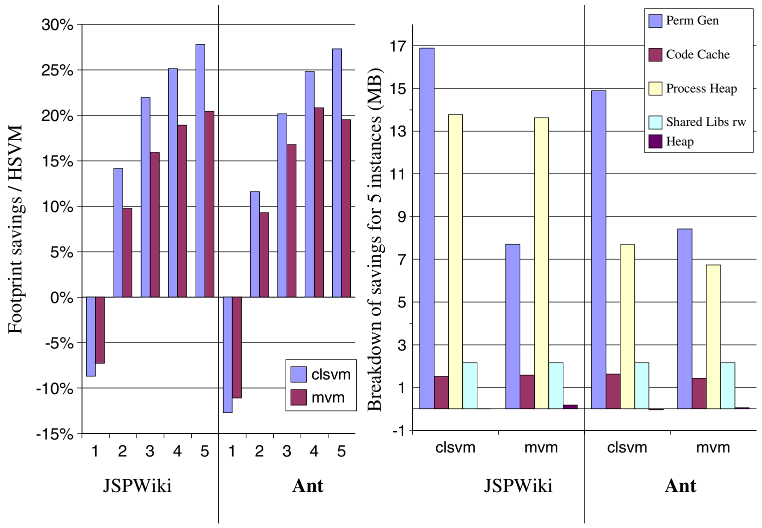


Fig. 3. Footprint savings for JSPWiki on Tomcat and for Ant. The left-hand part shows savings relatively to HSVM as the number of program instance grows. The right-hand part shows the breakdown of savings for 5 running programs

footprint reduction relative to HSVM; the right-hand part shows a breakdown of the savings for 5 instances of the measured programs.

When executing a single program, both MVM and CLSVM fare worse than HSVM, because the memory overhead of the login isolate is not amortized. This results in footprint increase ranging from 7.2% to 12.7%. The overhead for CLSVM is larger than for MVM, because of (i) the additional data structures required for loader re-entrance (e.g., the shared class repository, SHA digests), which are not amortized due to the absence of sharing; (ii), the extra overhead due to the classes used by CLSVM to compute SHA digests of class files (e.g., `java.security.MessageDigest`); and, (iii), the systematic decomposition of the runtime representation of *all* classes, including task re-entrant ones, into loader-dependent and loaded-independent parts, which, in the absence of loader re-entrant sharing, brings no benefits.

As soon as more than one application is executed, the benefits of sharing outweigh the overhead of the login isolate, and both MVM and CLSVM reduce the aggregate footprint of programs when compared to HSVM. The savings increase with the number of running programs, and increase faster with CLSVM than with MVM. The breakdown of savings shown in the right-hand part of Figure 3 offers insight into the reasons for this behavior. There are four sources of savings: the permanent generation (a garbage collected area that holds most of the runtime representation of classes); the code cache (where the compiled code produced by the dynamic compiler resides); the JVM's process C-heap (which stores some of the dynamic data structures of the JVM); and the private segments of the shared libraries used by the JVM.

Table 3. Population of classes for the programs used in the experiments

defining loader(s)	Ant	Tomcat	*db*	*javac*	*mpegaudio*	*jack*
Primordial	343	476	316	316	315	316
System	149	14	256	269	263	260
User-defined	540	806	9	149	57	60

The permanent generation and the process's heap are the main contributors to memory footprint reduction. Code cache can also contribute to savings, depending on the amount of re-entrant compiled methods. Sharing the runtime representation of classes across user-defined loaders mostly affects the permanent generation. For both Ant and Tomcat/JSPWiki, the permanent generation of CLSVM grows three times slower than that of MVM with each additional program instance. All other memory areas grow at almost the same pace.

The above demonstrates that CLSVM can bring benefits to these applications over and beyond what MVM already provides. CLSVM's gains, relative to MVM's, depend on the ratio of the number of classes defined by program-defined loaders (and their size) to the number of other classes. Both Ant and Tomcat/JSPWiki heavily exploit user-defined loaders: over 50% of classes are defined by them (see Table 3).

The bytecode transformation workload gives a more contrasted picture due to the much smaller proportion of classes defined by user-defined loaders[2]: across the SPECjvm98 programs, the population of user-defined classes (i.e., the classes subjected to runtime bytecode transformation) is between 4 (*javac*) to 60 (*db*) times smaller than the population of other classes. The bytecode transformation tool alone accounts for between 35% to 42% of the total population of loaded classes. Despite this the sharing across loaders of CLSVM brings visible benefits.

The left-hand part of Figure 4 compares the footprint savings from MVM and CLSVM for a sample of the SPECjvm98 programs. The sample is chosen for its differing footprint characteristics: *javac* (respectively, *db*) has the largest (respectively, smallest) population of classes defined by user-defined loaders; *megpaudio* and *jack* have almost the same population of loaded classes, but they vastly differ in terms of memory usage, as shown on the right-hand part of Figure 4: when run with HSVM, the heap in *jack* accounts only for 30% of the total footprint, compared to 50% for *mpegaudio*, and over 60% for both *javac* and *db*. As a result, the benefits of sharing are more pronounced for *jack* than for the other programs. Like before, the overhead of the login isolate is felt for the first program invocation, and erased as soon as more than one program is run. The footprint of *db* is initially worse with CLSVM than MVM because of the very small number of loader re-entrant classes. In this case, the overhead of the systematic de-

[2] The bytecode editing loader rewrites only the classes from the SPECjvm98 programs, not the JDK classes these programs use.

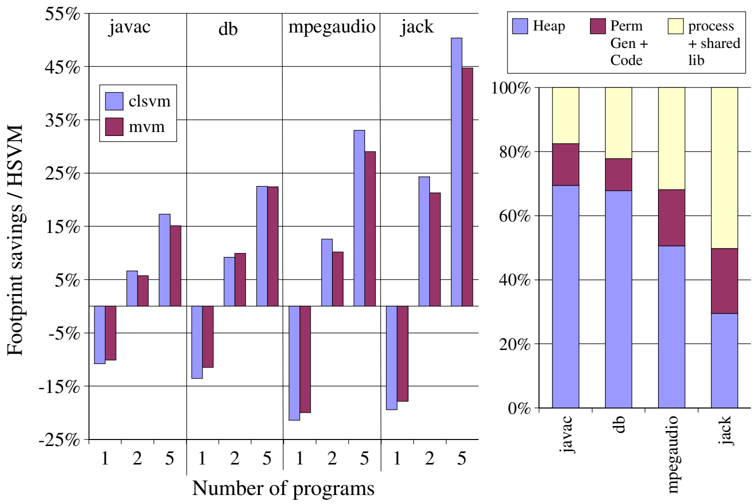


Fig. 4. Footprint savings for the bytecode transformation workload

composition of the runtime representation of classes into loader-dependent and loaded-independent parts slightly disadvantages CLSVM.

Likewise, when user-defined loaders are not used by programs, this decomposition adds an almost unnoticeable regression in footprint compared to MVM (less than 0.3% across all the SPECjvm98 programs).

5.3 Performance

One of our goals is to ensure that loader re-entrance does not negatively impact performance when user-defined class loaders are not used, or when there are no opportunities for sharing. The left-hand part of Figure 5 shows the performance improvement relative to HSVM on a sample of the SPECjvm98 benchmarks.

Both CLSVM and MVM improve performance noticeably when compared to HSVM. This is due to two factors. First, methods of classes defined by the primordial and system loaders are compiled into task re-entrant code that is shared across multiple programs, whether these execute concurrently or serially. Thus, programs benefit from the elimination of dynamic compilation and interpretation costs. Second, the invocation counters of methods are shared across program execution, which allows identifying and compiling more hot methods that what can be identified with a single program execution. As a result, a larger set of compiled methods is available for programs in MVM and CLSVM than in HSVM.

CLSVM slightly degrades performance compared to MVM on these benchmarks. We attribute this to the overhead of dealing with two method representations, which adds runtime tests to the calling convention of interpreted methods in CLSVM.

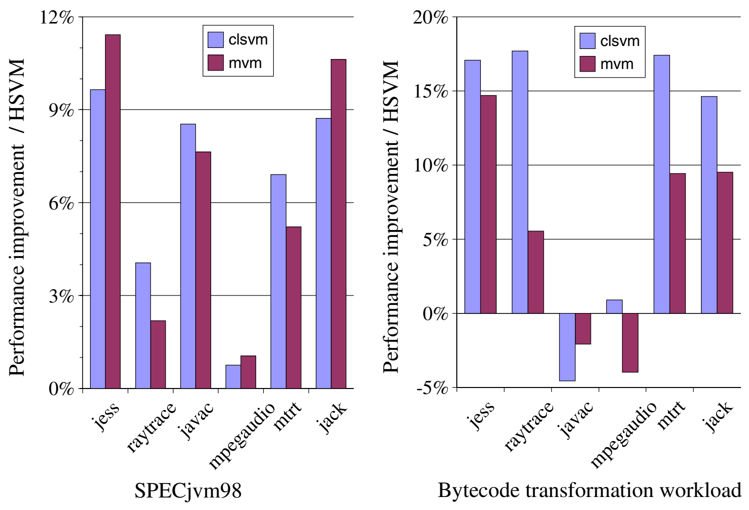


Fig. 5. Performance with respect to HSVM. The right-hand part shows performance improvement with programs from the SPECjvm98 programs. The left-hand part show performance improvement with the bytecode transformation workload

CLSVM further attempts to share compiled code across user-defined loaders or to produce new code by cloning existing code and changing its loader-dependent part. The effect of this strategy can be observed on the right-hand part of Figure 5, which shows the performance of the bytecode transformation workload applied to a sample of the SPECjvm98 suite. CLSVM's strategy for amortizing the cost of dynamic compilation across user-defined class loaders improves performance relatively to MVM. The impact depends on the proportion of loader-dependent compiled methods to the total number of compiled methods, and to the amount of loader-dependent code that can be shared. For example, 24% of compiled methods pertains to classes defined by user-defined loaders in *jess* and 30% for *raytrace*. However, only 12% of these are shared across loaders for *jess*, whereas nearly 45% are shared for *raytrace*. Thus *raytrace* benefits much more from CLSVM. The current prototype of CLSVM is not capable of sharing or cloning code that inlines loader re-entrant methods. In this case, the method needs to be recompiled. This limitation impacts directly *jess*, *javac*, and *mpegaudio*, since these have a substantial amount[3] of compiled loader re-entrant methods that inlines other loader re-entrant methods.

CLSVM outperforms MVM on *mpegaudio* because of differences in how static fields are accessed in both.

The frequency of *mpegaudio*'s accesses to static fields is about two orders of magnitute higher than for other SPECjvm98 benchmark. The difference in how static fields are accessed in both virtual machines explains why CLSVM

[3] 31%, 14% and 11% for, respectively, *jess*, *javac* and *mpegaudio*.

outperforms MVM on this benchmark, and also why MVM performs worse than HSVM. Specifically, MVM incurs the cost of an additional level of indirection when accessing static fields, regardless of whether the class defining the variable is task re-entrant or not.

Performance data obtained with Ant (not shown here) indicate 3.3% improvement of CLSVM over MVM, bringing performance relative to HSVM from 26.4% to 28.8%.

6 Related Work

Our work relates to recent efforts to share the main-memory runtime representation of classes between programs. This form of sharing can reduce both the footprint of Java programs and factor out the runtime costs of transforming class file to an optimized, architecture-specific class type representation.

One approach to sharing consists of launching for each Java program a separate OS process to execute an instance of the JVM, and to store the sharable part of the runtime representation of classes in an area of memory shared by the JVM processes [6, 8]. What is made sharable varies according to implementations: at the minimum, the bytecodes of methods, which are by far the largest part of the runtime representation of classes are shared[4]. The immutable part of the runtime representation of classes, such as symbol constants and some of the meta-description of methods and fields, can also be shared [6]. Additional information necessary to reconstruct the mutable part of the runtime representation of a class can also be stored in the shared area, to avoid parsing the original class file when constructing the process-private part of the class's representation. ShMVM-C [6] goes one step further by also making the output of the dynamic compiler program re-entrant so that multiple JVM processes can share it.

An alternative to storing the sharable part of a class in shared memory is to encode the whole runtime representation of classes in a binary format natively supported by the host OS's shared libraries mechanism. For instance, SLVM [19] encodes the main-memory representation of classes in the ELF format. The resulting binaries are relocatable. Loading and relocation are performed by the linker. Better memory utilization can result from this approach, due to the systematic copy-on-write policy implemented by the linker on any page of mutable sections of the shared library.

Another approach to transforming Java classes into shared libraries is exemplified by GCJ [9], a portable, optimizing, ahead-of-time compiler for the Java programming language. The run-time system of GCJ supports program-defined class loaders and dynamic class loading, but code loaded this way can only be interpreted. [20] describes an extension to GCJ that supports sharing of code compiled ahead of time across program even if the class is not linked to classes

[4] The implementation of class data sharing in the Java HotSpot Virtual machine 1.5.0 is a variant of this, where method bytecodes of boot classes are stored in a file that is memory-mapped at program startup [17].

with the same definition in each program. To this end [20] re-introduces runtime functions and data structures commonly found in standard virtual machine (e.g., vtables are constructed at runtime and each class is associated with a table of links to external symbols that get filled up at class load-time).

MVM [5] tackles the memory footprint problem by collocating multiple Java programs in the same OS process, and executing them within a single JVM capable of multi-tasking. Most of the runtime representation of classes, including compiled code, is made program re-entrant and shared across all tasks. Interference among programs is prevented by replicating the program-dependent part of the runtime representation of classes.

None of the systems mentioned above is capable of sharing, either within the same program or across programs, the runtime representation of classes defined by arbitrary user-defined class loaders, especially when they edit bytecodes or generates class file at runtime.

Like MVM and CLSVM, Microsoft's .NET allows for isolated execution of multiple applications in the same process [14]. The platform can be configured to transparently share some meta-data, although not as aggressively as CLSVM. Decisions concerning the trade-off between memory footprint and performance can be made at deployment time. *Domain-neutral* deployments result in slower execution as non-static meta-data is shared while static data and static code are replicated. The additional logic that directs callers to the appropriate static code or data is thus needed to provide application isolation. The standard form of application deployment in .NET does not have this feature, thus potentially providing better performance at the expense of memory footprint.

QuickSilver [15] amortizes the cost of producing a high-performance runtime image of a program by compiling the methods of a class into a relocatable format and storing the result of the compilation in files. Compiled method files can be generated off-line, or when an instance of the JVM exits. Subsequent executions of the program load the compiled method files, if available, upon the class definition. Before its use, the compiled code is subjected to validation tests that determine if the code can be re-used by the running program. If validation succeeds, the code is *"stitched"* according to the state of the running JVM. Otherwise, the code undergoes standard dynamic compilation. In its latest version [12], QuickSilver generates code that uses an indirection mechanism in order to make most of the compiled code of methods read-only, thus reducing the aggregate footprint of multiple JVM instances. However, the indirection mechanism makes the code dependent on the address of an indirection table specific to one class loader. As a result, code cannot be shared across multiple class loaders within the same JVM instance. This limits the usefulness of this approach, especially for server environments where class loader-based containers are often used.

7 Conclusions

Defining class loading policies is a commonly used feature of the Java programming language. The reliance on class loaders is likely to grow, partly due to a growing popularity of load-time bytecode transformations applicable to aspect-oriented programming. However, existing implementations of the JVM poorly support class loaders with regard to resource utilization.

This paper describes CLSVM, a multi-tasking implementation of the JVM capable of transparently sharing the runtime representation of classes, including their bytecodes and compiled code, across multiple defining loaders. Sharing is achieved by separating out the part of the runtime representation of a class that depends on symbolic link resolution and by making bytecode interpretation loader re-entrant. Re-entrance is implemented by the addition of class resolution and initialization barriers and by efficient access to loader-dependent components of the runtime representation of classes. The dynamic compiler exploits sharing by maintaining loader dependencies and using them to determine when sharing of compiled methods across loaders is possible. If the required sharing conditions are not met, the code is cloned to avoid its compilation from bytecodes.

The presented techniques enhance MVM so as to extend the scope and benefits of code sharing to classes defined by arbitrary class loaders, regardless of whether these loaders pertain to different programs or the same one. Further, classes subjected to bytecode transformation at runtime also benefit from sharing.

The impact of the mechanisms implemented by CLSVM on end-to-end application performance is highly dependent on the proportion of compiled loader re-entrant methods that can be shared. Performance relative to MVM varies between −3% to +12.8%, the highest improvements corresponding to cases when sharing of loader-reentrant compiled methods can be exploited. When compared to the Java HotSpot virtual machine, the gains are between 0.9% to 17%. Application start-up time is almost identical for both CLSVM and MVM, and between 1.85 (command-line launching of applications) and 26 (programmatic launching of applications) times faster than for the Java HotSpot virtual machine.

Along with not degrading start-up time and bringing about a modest improvement in end-to-end performance compared to MVM, the main motivation for this work, was to decrease memory footprint of applications that exploit user-defined class loading. This goal has been achieved: for example, for applications like Ant and Tomcat CLSVM improves the memory savings already achieved by MVM by between 15% to 40%, bringing total memory footprint down by 11.6% to 28% with respect to the Java HotSpot virtual machine. When class loading mechanisms are not used, memory overhead relative to MVM remains below 3%.

References

1. http://java.sun.com/products/jsp.
2. http://www.jspwiki.org.
3. J. Arnold. Shared Libraries on UNIX System V. In *Summer USENIX Conference*, Atlanta, GA, 1986.
4. D. Balfanz and L. Gong. Experience with Secure Multi- Processing in Java. Technical Report 560-97, Department of Computer Science, Princeton University, Sept. 1997.

5. G. Czajkowski and L. Daynès. Multitasking without Compromise: A Virtual Machine Evolution. In *ACM OOPSLA'01*, Tampa, FL, Oct. 2001.
6. G. Czajkowski, L. Daynès, and N. Nystrom. Code Sharing among Virtual Machines. In *ECOOP'02*, Malaga, Spain, June 2002.
7. G. Czajkowski, L. Daynès, and B. Titzer. A multi-user virtual machine. In *USENIX*, San Antonio, TX, 2003.
8. D. Dillenberger, R. Bordawekar, C. W. Clark, D. Durand, D. Emmes, O. Gohda, S. Howard, M. F. Oliver, F. Samuel, and R. W. S. John. Building a Java^{TM} Virtual Machine for Server Applications: The JVM on OS/390. *IBM Systems Journal*, 39(1), 2000.
9. Free Software Foundation (FSF). GCJ: The GNU Compiler for Java., 2003.
10. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java^{TM} Language Specification*. The Java^{TM} Series. Addison Wesley, second edition edition, Sept. 2000.
11. Java Community Process. JSR 121: Application Isolation API., 2003.
12. P. G. Joisha, S. P. Midkiff, M. J. Serrano, and M. Gupta. A framework for efficient reuse of binary code in java. In *International Conference on Supercomputing*, pages 440–453, 2001.
13. S. Liang and G. Bracha. Dynamic Class Loading in the Java Virtual Machine. In *ACM OOPSLA'98*, Oct. 1998.
14. Microsoft Corp. *Programming with Application Domains and Assemblies.* http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconprogrammingwithapplicationdomainsassemblies.asp, 2005.
15. M. J. Serrano, R. Bordawekar, S. P. Midkiff, and M. Gupta. Quicksilver: a Quasi-Static Compiler for Java. In *ACM OOPSLA'00*, Oct. 2000.
16. Sun Microsystems Inc. The Java HotSpot Performance Engine Architecture. http://java.sun.com/products/hotspot/whitepaper.html, 1999.
17. Sun Microsystems Inc. Class Data Sharing. http://java.sun.com/j2se/1.5.0/docs/guide/vm/class-data-sharing.html, 2004.
18. US Department of Commerce. Secure hash standard, Apr. 1995.
19. B. Wong, G. Czajkowski, and L. Daynès. Dynamically loaded classes as shared libraries. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Nice, France, 2003.
20. D. Yu, Z. Shao, and V. Trifonov. Supporting binary compatibility with static compilation. In *2nd Java Virtual Machine Research and Technology Symposium (JVM'02)*, pages 165–180, 2002.

Aspect-Oriented Programming Beyond Dependency Injection

Shigeru Chiba and Rei Ishikawa

Dept. of Mathematical and Computing Sciences,
Tokyo Institute of Technology

Abstract. Dependency injection is a hot topic among industrial developers using component frameworks. This paper first mentions that dependency injection and aspect-oriented programming share the same goal, which is to reduce dependency among components for better reusability. However, existing aspect-oriented programming languages/ frameworks, in particular, AspectJ, are not perfectly suitable for expressing inter-component dependency with a simple and straightforward representation. Their limited kinds of implicit construction of aspect instances (or implementations) cannot fully express inter-component dependency. This paper points out this fact and proposes our aspect-oriented programming system named *GluonJ* to address this problem. GluonJ allows developers to explicitly construct and associate an aspect implementation with aspect targets.

1 Introduction

A key feature of the new generation of component frameworks like the Spring framework [10] is dependency injection [6]. It is a programming technique for reducing the dependency among components and thereby improving the reusability of the components. If a component includes sub-components, reusing only that component *as is* independently of those sub-components is often difficult. For example, if one of those sub-components is for accessing a particular database, it might need to be replaced with another sub-component for a different database when the component is reused. The original program of that component must be edited for the reuse since it includes the code for instantiating the sub-component. The idea of dependency injection is to move the code for instantiating sub-components from the program of a component to a component framework, which makes instances of sub-components specified by a separate configuration file (usually an XML file) and automatically stores them in the component.

Dependency injection is a good idea for reducing inter-component dependency. However, since existing component frameworks with dependency injection are implemented with a normal language, mostly in Java, the independence and reusability of components are unsatisfactory. For example, the programs of components depend on a particular component framework and thus they must be modified when they are reused with a different framework.

This paper mentions that dependency injection and aspect-oriented programming (AOP) share the same goal from a practical viewpoint. Hence introducing the ideas of aspect-oriented programming into this problem domain provides us with better ability for reducing dependency among components. However, existing aspect-oriented systems used with component frameworks are mostly based on the architecture of AspectJ [12] and thus their design has never been perfectly appropriate for reducing inter-component dependency. In fact, aspect-oriented programming and dependency injection have been regarded as being orthogonal and used for different applications and purposes. Otherwise, aspect-oriented programming is just an implementation mechanism of dependency injection.

This paper presents our aspect-oriented programming framework named *GluonJ*, which we designed for dealing with dependency among components in Java. Although the basic design of GluonJ is based on that of AspectJ, GluonJ allows developers to explicitly associate an aspect implementation with aspect targets. The aspect implementation is a component implementing a crosscutting concern and the aspect targets are components that the concern cuts across. Existing aspect-oriented systems only allow implicit association and hence they cannot fully express inter-component dependency as an aspect.

The organization of the rest of this paper is followings. In Section 2, we discuss dependency injection and problems of the current design. Section 3 presents our aspect-oriented programming framework named GluonJ. Section 4 mentions comparison between GluonJ and AspectJ. Section 5 briefly describes related work and Section 6 concludes this paper.

2 Loosely Coupled Components

This section first overviews the idea of dependency injection. Then it mentions that dependency injection makes components dependent on a particular component framework and a naive aspect-oriented solution is not satisfactory.

2.1 Dependency Injection

Dependency injection enables loosely-coupled components, which are thereby highly reusable. If a component contains a sub-component, it will be usually difficult to reuse without the sub-component since these two components will be tightly coupled. For example, suppose that the program of that component is as following (Figure 1):

```
public class MyBusinessTask {
  Database db;

  public MyBusinessTask() {
    db = new MySQL();
  }
  public void doMyJob() {
    Object result = db.query("SELECT USER.NAME FROM USER");
    System.out.println(result);
  }
}
```

Note that this component contains a MySQL object as a sub-component. MySQL is a class implementing a Database interface. Since MyBusinessTask is tightly coupled with MySQL, the constructor of MyBusinessTask must be modified if MyBusinessTask is reused with another database accessor, for example, a PostgreSQL object. The new constructor would be:

```
public MyBusinessTask() {
  db = new PostgreSQL();    // not new MySQL()
}
```

Dependency injection loosens the connection between MyBusinessTask and MySQL. It enables us to reuse MyBusinessTask without modification even if we must switch a database accessor from MySQL to PostgreSQL. The program of MyBusinessTask would be changed into this:

```
public class MyBusinessTask {
  Database db;

  public void setDb(Database d) {
    db = d;
  }

  public void doMyJob() {
    Object result = db.query("SELECT USER.NAME FROM USER");
    System.out.println(result);
  }
}
```

Now, no constructor in MyBusinessTask initializes the value of the db field. It is initialized (*or injected*) by a factory method provided by the framework

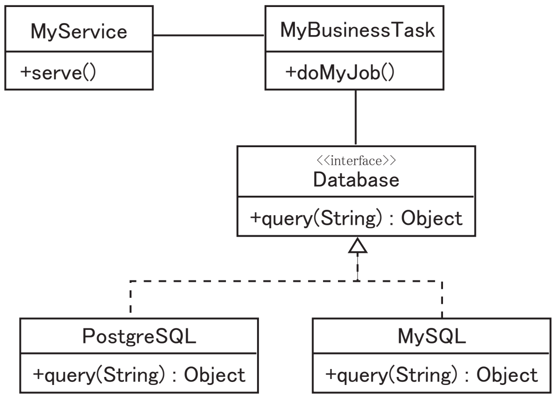


Fig. 1. Class diagram for our example scenario

supporting dependency injection. Thus, a MyBusinessTask object must not be constructed by the new operator but the factory method (or, otherwise, a MyBusinessTask object constructed by the new operator must be explicitly passed to a method provided by the component framework for performing dependency injection). For example, the code snippet below constructs a MyBusinessTask object:

```
XmlBeanFactory factory = new XmlBeanFactory(
    new InputStreamResource(new FileInputStream("beans.xml")));
MyBusinessTask myTask = (MyBusinessTask)factory.getBean("myTask");
```

Here, XmlBeanFactory is a factory class provided by a component framework. The getBean method constructs an instance of MyBusinessTask and initializes the value of the db field. It constructs a MySQL object and assigns it to the db field. This initialization is executed according to an XML configuration file beans.xml. The parameter to getBean is a key to find a configuration entry for MyBusinessTask in beans.xml.

Reusing MyBusinessTask with not MySQL but PostgreSQL is easy. We do not have to modify the program of MyBusinessTask but we have only to modify the configuration file beans.xml, which specifies how the db field is initialized. According to the configuration file, the getBean method will construct a PostgreSQL object and assign it to the db field.

However, using a factory method is annoying. Furthermore, if the hierarchy of components is more complicated, the program of the components depends on a particular component framework. Suppose that MyBusinessTask is a subcomponent of another component MyService. The program of MyService would be something like this:

```
public class MyService {
  MyBusinessTask task;

  public MyService(XmlBeanFactory factory) {
    task = factory.getBean("myTask");
  }

  public void serve() {
    task.doMyJob();
  }
}
```

MyService and MyBusinessTask do not require to be modified when they are reused with either MySQL or PostgreSQL. Only the configuration file must be modified.

However, the program above includes XmlBeanFactory, which is a class provided by the component framework. MyService and MyBusinessTask, therefore, depend on the component framework. We cannot reuse them *as is* with another component framework. If we switch component frameworks, we also have to modify the MyService class. This problem can be avoided if we also construct a MyService object through an XmlBeanFactory object. Since the component framework constructs a MyBusinessTask object for injecting it into the task field in a MyService object, the constructor of MyService does not have to explicitly call the getBean method on factory. We can write the program of MyService without referring to XmlBeanFactory. However, this solution requires all components to be constructed through an XmlBeanFactory object. For example, MyService might be always reused with MyBusinessTask since these two components are tightly coupled. If so, dependency injection is not necessary for MyService but we must write a configuration file for MyService and construct the MyBusinessTask component through an XmlBeanFactory object. This programming convention is somewhat awkward.

2.2 Aspect-Oriented Programming

The programming problem of dependency injection mentioned above is that the programs of components depend on a particular component framework; we cannot switch component frameworks without modifying the programs of the components. We must stay with a particular component framework. As we develop a larger collection of useful components, switching component frameworks becomes more difficult.

This problem can be easily solved if we accept aspect-oriented programming. Since the source of this problem is that we cannot intercept object construction within confines of regular Java, we can solve the problem by using aspect-oriented programming for intercepting object creation. For example, if we use AspectJ, we can intercept construction of MyBusinessTask by the following program:

```
privileged aspect DependencyInjection {
  after(MyBusinessTask s):
      execution(void MyBusinessTask.new(..)) && this(s) {
    s.db = new MySQL();
  }
}
```

This aspect corresponds to an XML configuration file of the component framework shown in the previous subsection. If we define this aspect, the definition of MyService can be written without a factory method:

```
public class MyService {
  MyBusinessTask task;

  public MyService() {
    task = new MyBusinessTask();
  }

  public void serve() {
    task.doMyJob();
  }
}
```

If this class is compiled with the aspect, the construction of MyBusinessTask is intercepted and then a MySQL object is assigned to the db field in MyBusinessTask.

Although this solution using AspectJ requires our development environments to support a new language — AspectJ, we can use a Java-based aspect-oriented framework such as JBoss AOP[9] and AspectWerkz [1] if we want to stay with regular Java.

The solution with aspect-oriented programming enables reusable components that are even independent of component frameworks. We can switch component frameworks and aspect-oriented programming systems without modifying the definition of MyService. Only the DependencyInjection aspect must be rewritten if the aspect-oriented programming system is changed.

2.3 Is This Really a Right Solution?

Although AspectJ could make a MyService component independent of a component framework, this solution would not be a good example of aspect-oriented programming. This solution uses AspectJ only as an implementation mechanism for intercepting object construction. It can be implemented with not only AspectJ but another mechanism such as a metaobject protocol [11, 3, 7, 20], which also enables intercepting object construction. In fact, the description in the DependencyInjection aspect does not directly express the dependency relation among components. It is procedural and includes implementation details. The level of abstraction is relatively low.

However, we mention that aspect-oriented programming is a right approach to solve the problem illustrated above, and more generally, to reduce dependency among components. In other words, aspect-oriented programming and dependency injection share the same goal, which is to reduce inter-component dependency for better component reusability. Aspect-oriented programming is known as a paradigm for implementing a crosscutting concern as an independent and separated component. A concern is called crosscutting if the implementation of that concern in a non aspect-oriented language is tangled with the implementation of other components. Another interpretation of this definition is that aspect-oriented programming is a paradigm for separating tightly coupled components so that they will be less dependent on each other and hence easily reusable. This is the same goal of dependency injection although dependency injection can reduce only a particular kind of dependency while aspect-oriented programming covers a wider range of dependency.

Unfortunately, existing aspect-oriented programming systems represented by AspectJ are not perfectly suitable to reduce inter-component dependency. A main problem is that they implicitly associate an aspect implementation with an aspect target. Here, the aspect implementation is a component implementing a crosscutting concern and the aspect target is a component that the concern cuts across. If program execution reaches a join point specified by a pointcut, an advice body is executed on the aspect implementation associated with the aspect target including that join point. In AspectJ, an aspect implementation is not a regular Java object but an instance of aspect. Thus we cannot use an existing component written in Java as an aspect implementation. To avoid this problem, several frameworks such as JBoss AOP and AspectWerkz allow using a regular object as an aspect implementation. Pointcuts are described in an aspect-binding file, that is, an XML file. However, such a regular object is *implicitly* constructed and associated with the aspect target. An aspect instance of AspectJ is also implicitly constructed and associated.

This implicit association between an aspect implementation and an aspect target has two problems. First, expressing dependency injection is made difficult. Dependency injection can be regarded as associating a component with another. An injected component corresponds to an aspect implementation. If developers do not have full control of the association, they cannot naturally express dependency injection with aspect-oriented programming.

The other problem is that the implicit association does not provide sufficient expressive power enough to express various relations of inter-component dependency as aspects. Although AspectJ lets developers select a scheme from issingleton, perthis, and so on, these options cover only limited kinds of relations among components. Developers might want to associate an aspect implementation with a group of aspect targets. AspectJ does not support this kinds of association. The relations among components in general do not always form a simple tree structure. Hence an aspect implementation is not always a sub-component owned by a single aspect target. It may be referred to by several different aspect targets. It may be a singleton and hence shared among all aspect targets. Existing aspect-oriented programming systems allow only selecting from limited types of the relation and they implicitly construct an aspect implementation and associate it with the aspect target according to the selected option. Therefore, developers must often redesign the relations among components so that the relations fit one of the types provided by the system.

3 GluonJ

This section presents *GluonJ*, which is our new aspect-oriented programming framework for Java. The design of GluonJ is based on the pointcut-advice architecture of AspectJ. However, this architecture has been restructured for GluonJ to provide a simpler programming model for reducing inter-component dependency.

GluonJ separates aspect bindings from aspect implementations. Aspect implementations are regular Java objects, which implement a crosscutting concern. They corresponds to an aspect instance in AspectJ. Aspect binding is the *glue* code described in XML. It specifies how an aspect implementation is associated with aspect targets, which the aspect implementation cuts across, at specified join points. The aspect binding includes not only pointcuts but also code fragments written in Java. These code fragments *explicitly* specify which aspect implementation is associated with the aspect targets. If program execution reaches a join point specified by a pointcut, then the code fragment is executed. It can explicitly construct an aspect implementation and call a method on that aspect implementation to execute a crosscutting concern. Since GluonJ was designed for reducing inter-component dependency, GluonJ lets developers to describe

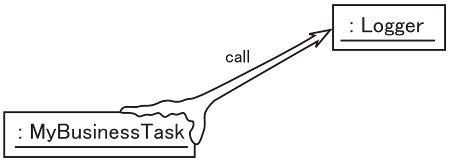


Fig. 2. The aspect of GluonJ is *glue*, which connects two components. Unlike the aspect of AspectJ, the aspect of GluonJ is not part of the Logger component or the MyBusinessTask component

the code fragments in the aspect binding to explicitly express various relations between aspect targets and aspect implementations.

3.1 Logging Example

To illustrate the usage of GluonJ, we below show the implementation of a logging concern in GluonJ. The logging concern is a well-known crosscutting concern, which is often used for showing the usage of an aspect-oriented programming system. The goal of this example is to extend the behavior of MyBusinessTask so that a log message will be printed when a method in MyBusinessTask is executed. However, we cannot modify the program of MyBusinessTask for this extension since modifying that program means that MyBusinessTask includes part of the implementation of the logging concern. The logging concern must be implemented as an independent component separated from the other components.

In GluonJ, we first define a Logger class in Java:

```
public class Logger {
  public void log() {
    System.out.println("method execution");
  }
}
```

Logger is a class for the logging concern. Unlike AspectJ, GluonJ uses a regular Java object as an aspect implementation, which is a component implementing a crosscutting concern such as the logging concern.

In GluonJ, an aspect means the aspect binding written in XML, for example, for describing the dependency between a Logger object and other objects. It glues a Logger object to the objects that must produce log messages (Figure 2). The aspect does not include an aspect implementation, which is the Logger class. For example, the following aspect specifies that a log message is printed just after a method in MyBusinessTask is executed:

```
<aspect>
  <injection>
    Logger MyBusinessTask.aspect = new Logger();
  </injection>
  <advice>
    <pointcut>
```

```
      execution(* MyBusinessTask.*(..))
    </pointcut>
    <after>
      Logger.aspectOf(this).log();
    </after>
  </advice>
</aspect>
```

This aspect makes it possible to keep the two components MyBusinessTask and Logger loosely coupled with low dependency on each other. The GluonJ compiler automatically transforms the program of MyBusinessTask according to this aspect at compilation time. Thus, we can change the behavior of MyBusinessTask without manually modifying the program of MyBusinessTask.

The statement surrounded with the injection tag specifies a connection between a MyBusinessTask object and a Logger object. It means that, when a MyBusinessTask is constructed, a Logger object is also constructed and then associated with that MyBusinessTask object. The syntax of this statement is the same as the intertype field declaration in AspectJ except aspect is not a field name but a special keyword.

The elements surrounded with the advice tag are pointcut and after advice. The pointcut is surrounded with the pointcut tag. It is the almost same language element as AspectJ's pointcut except the syntax. In the aspect shown above, the pointcut picks out as join points method execution on MyBusinessTask objects. The code snippet surrounded with the after tag is an advice body, which is executed just after a thread of control reaches the execution point specified by the pointcut. The code snippet is written in regular Java except that a special form aspectOf is available in that code snippet. In the aspect shown above, Logger.aspectOf(this) is used to obtain the Logger object associated with the MyBusinessTask object referred to by this. aspectOf is a special form that is used in the following form:

<*class name*>`.aspectOf(`<*object*>`)`

This special form is used to obtain an object associated with another object by the injection tag. It returns the object that is of the <*class name*> type and is associated with the given <*object*>.

The advice body, which is the code snippet surrounded with the after tag, is executed in the context of the join point picked out by a pointcut. In the case of our example, the advice body is executed on an MyBusinessTask object since the join points picked out are the execution points when a method is executed on that object. Therefore, this appearing in the advice body refers to that MyBusinessTask object although it refers to an aspect instance in AspectJ. If needed, the advice body can access private fields and methods in MyBusinessTask. This is not allowed in AspectJ unless the aspect is privileged. On the other hand, the advice body in GluonJ cannot access private fields or methods in Logger. The visibility scope is determined by the execution context of the advice body. In AspectJ, it is an instance of the aspect while it is the same context as the join point in GluonJ.

A unique feature of GluonJ is that an aspect implementation must be explicitly constructed in the aspect. In our example, a Logger object was constructed in the statement surrounded with the injection tag. Then it is associated with the MyBusinessTask object and used in the advice body. If program execution reaches a join point specified by a pointcut, the advice body is executed and it explicitly calls a method on the associated aspect implementation, that is, the Logger object. Note that GluonJ never instantiates an aspect since the aspect is glue code in GluonJ. From the implementation viewpoint, the code snippet in the aspect is merged into the methods of the aspect target, that is, MyBusinessTask.

3.2 Using the injection Tag for Dependency Injection

An advice body in GluonJ can be any Java code. It does not have to call aspectOf. For example, if a MyBusinessTask object had a field and that field refers to a Logger object, an advice body could call the log method on the object referred to by that field instead of the object returned by aspectOf.

The special form aspectOf and the injection tag are provided for adding a new field to an existing class while avoiding naming conflict. An aspect can give a specific name to an added new field, for example, by the following description:

```
<injection>
  Logger MyBusinessTask.link = new Logger();
</injection>
```

This adds a new field named link to the MyBusinessTask class and it initializes the value of that field so that it refers to a Logger object. The type of that field is Logger. However, this may cause naming conflict if another aspect adds a link field to the MyBusinessTask class.

If a special keyword aspect is specified as the name of the added field, this field becomes an anonymous field, that is, a field that has no name. An anonymous field can be accessed only through the special form aspectOf. For example, Logger.aspectOf(p) represents the anonymous field that is Logger type and belongs to the object p. We do not have to manually choose a unique field name for avoiding naming conflict.

There is also another rule with respect to the name of a newly added field. If the specified field name is the same as an already existing field in the same class, a new field is never added to the class. The initial value specified in the block surrounded with injection is assigned to that existing field with the same name.

This rule allows us to describe dependency injection with a simple aspect. For example, the example shown in the previous section can be described with the following aspect:

```
<aspect>
  <injection>
    Database MyBusinessTask.db = new MySQL();
  </injection>
</aspect>
```

This aspect specifies that a MySQL object is constructed and assigned to the db field in MyBusinessTask when an MyBusinessTask object is constructed. Since the db field already exists, no new field is added to MyBusinessTask. The aspect does not have to include a pointcut for picking out the construction of a MyBusinessTask object.

Although the block surrounded with the injection tag is similar to the intertype field declaration of AspectJ, it is not the same language element as the intertype field declaration. The added fields in GluonJ are private fields only accessible in the class to which those fields are added. On the other hand, private fields added by intertype field declarations of AspectJ are not accessible from the class to which those fields are added. They are only accessible from the aspect (implementation) that declares those fields.

3.3 Dependency Reduction

GluonJ was designed particularly for addressing inter-component dependency, which is a common goal to aspect-oriented programming and dependency injection. Thus GluonJ provides mechanisms for dealing with the two sources of the dependency: connections and method calls among components.

A component depends on another component if the former has a connection to the latter (i.e. the former has a reference to the latter) and/or the former calls a method on the latter. This dependency becomes a problem if the latter component implements a crosscutting concern. Let us call the former component *the caller* and the latter one *the callee.* In the example in Section 3.1, the caller is MyBusinessTask and the callee is Logger.

The inter-component dependency makes it difficult to reuse the caller-side component *as is.* If the callee is a crosscutting concern, it is not a sub-component of the caller; it is not contained in the caller or invisible from the outside of the caller. Therefore, those components should be independently reused without each other. For example, since Logger is a crosscutting concern and hence it is not crucial for implementing the function of MyBusinessTask, MyBusinessTask may be reused without Logger. Reusing the callee without the caller is easy; the program of that component can be reused *as is* for other software development. On the other hand, in regular Java, reusing the caller without the callee, for example, reusing MyBusinessTask without Logger needs to edit the program of the caller-side component MyBusinessTask since it includes method calls to the callee. These method calls must be eliminated from the program before the component is reused.

Connections Among Components: For reducing dependency due to connections among components, GluonJ provides the block surrounded with the injection tag. Although this dependency can be reduced with the technique of dependency injection, GluonJ enables framework independence discussed in Section 2.2 since it is an aspect-oriented programming (AOP) system. Furthermore, GluonJ provides direct support for expressing this dependency although in other AOP systems this dependency is indirectly expressed by advice intercepting object creation. We adopted this design of GluonJ because addressing the dependency due to inter-component connections is significant in the application domain of GluonJ.

Method Calls Among Components: For reducing dependency due to method calls, GluonJ provides the pointcut-advice architecture. For example, as we showed in Section 3.1, the dependency between MyBusinessTask and Logger due to the calls to the log method can be separately described in the block surrounded with the advice tag. This separation makes the method calls *implicit* and *non-invasive* and thus MyBusinessTask will be reusable independently of Logger. The reuse does not need editing the program.

Note that the method call on the Database object within the body of MyBusinessTask in Section 2.1 does not have to be implicit by being separately described in XML. This call is a crucial part of the function of MyBusinessTask and hence MyBusinessTask will never be reused without a component implementing the Database interface. We do not have to reduce the dependency due to this method call.

Since the pointcut-advice architecture of GluonJ was designed for reducing dependency due to method calls, the aspect implementation that a method is called on is explicitly specified in the advice body written in Java. That aspect implementation can be any regular Java object. It can be an object constructed in the block surrounded with injection but, if needed, it can be any other object. It is not flexible design to enable calling a method only on the aspect implementation that the runtime system implicitly constructs and associates with the aspect target. We revisit this issue in Section 4.

3.4 The Tags of GluonJ

A block surrounded with the aspect tag may include blocks surrounded with either the injection tag or the advice tag. We below show brief overview of the specifications of these tags.

Injection Tag: In a block surrounded with the injection tag, an anonymous field can be declared. For example, the following declaration adds a new anonymous field to the MyBusinessTask class:

```
<injection>
  Logger MyBusinessTask.aspect = new Logger(this);
</injection>
```

The initial value of the field is computed and assigned right after an instance of MyBusinessTask is constructed. The expression computing the initial value can be any Java expression. For example, it can include the this variable, which refers to that MyBusinessTask object in the example above.

If the declaration above starts with static, then a static field is added to the class. The initial value is assigned when the other static fields are initialized.

The field added by the declaration above is accessible only in the aspect. To obtain the value of the field, the special form aspectOf must be called. For example, Logger.aspectOf(t) returns the Logger object stored in the anonymous field of the MyBusinessTask object specified by t. If the anonymous field is static, then the parameter to aspectOf must be a class name such as MyBusinessTask.

A real name can be given to a field declared in an injection block. If an valid field name is specified instead of aspect, it is used as the name of the added field. That field can be accessed with that name as a regular field in Java. If there already exists the field with that specified name, a new field is not added but only the initial value specified in the injection block is assigned.

An anonymous field can be added to an object representing a control flow specified by the cflow pointcut designator. This mechanism is useful to obtain similar functionality to a percflow aspect instance in AspectJ. To declare such a field, the aspect should be something like this:

```
<injection>
  Logger Cflow(call(* MyBusinessTask.*(..)).aspect
      = new Logger();
</injection>
```

An anonymous field is added to an object specified by Cflow. It represents a control flow from the start to the end of the execution of a method in MyBusinessTask. It is automatically created while the program execution in that control flow. To obtain the value of this anonymous field, aspectOf must be called with the thisCflow special variable. For example,

```
Logger.aspectOf(thisCflow).log();
```

aspectOf returns the Logger object stored in thisCflow. thisCflow refers to the Cflow object representing the current control flow.

An anonymous field can be used to associate a group of objects with another object. This mechanism provides similar functionality to the association aspects [17]. For example,

```
<injection>
  Logger MyBusinessTask.aspect(Session) = new Logger(this, args);
</injection>
```

This declaration associates multiple Logger objects with one MyBusinessTask. this and args are special variables. These Logger objects are identified by a Session object given as a key. The type of the key is specified in the parentheses following aspect. Multiple keys can be specified. The associated objects are obtained by aspectOf. For example,

```
Logger.aspectOf(task, session).log();
```

This statement calls the log method on the Logger object associated with a combination of task and session. aspectOf takes two parameters: the first parameter is a MyBusinessTask object and the second one is a Session object. aspectOf returns an object associated with the combination of these objects passed as parameters. If any object has not been associated with the given combination, aspectOf constructs an object and associates it with that combination. In other words, an associated object is never constructed until aspectOf is called. In the case of the example above, a Logger object is constructed with parameters this and args. this refers to the first parameter to AspectOf (*i.e.* the MyBusinessTask object) and args refers to an array of Object. The elements of this array are the parameters to aspectOf except the first one. In this example, args is an array containing only the Session object as an element.

Advice Tag: A block surrounded with the advice tag consists of a pointcut and an advice body. The pointcut is specified by the pointcut tag. The syntax of the pointcut language was borrowed from AspectJ although the current implementation of GluonJ does not support the if and adviceexecution pointcut designators. Although && and || must be escaped, AND and OR can be used as substitution. The current implementation of GluonJ has neither supported a named pointcut. A pointcut parameter is defined by using the param tag. For example, the following aspect uses an int parameter i as a pointcut parameter. It is available in the pointcut and the advice body.

```
<advice>
  <param><name>i</name><type>int</type></param>
  <pointcut>
    execution(* MyBusinessTask.*(..)) AND args(i)
  </pointcut>
  <after>
    Logger.aspectOf(this).log(i);
  </after>
</advice>
```

An advice body can be before, after, or around. It is executed before, after, or around the join point picked out by the pointcut. Any Java statement can be specified as the advice body although the < and > operators must be escaped since an advice body is written in an XML file. A few special forms aspectOf(), thisCflow, and thisJoinPoint are available in the advice body. The thisCflow variable refers to a Cflow object representing the current control flow. The thisJoinPoint variable refers to an object representing the join point picked out by the pointcut. If the proceed method is called on thisJoinPoint, it executes the original computation at the join point. The return type of proceed() is Object. The proceed method is only available with around advice.

Reflection: Although aspectOf is available only in a advice body, GluonJ provides a reflection mechanism [18] for accessing anonymous fields from regular Java objects. Table 1 lists the static methods declared in Aspect for reflective accesses.

4 Comparison to AspectJ

Although GluonJ has borrowed a number of ideas from AspectJ, there are a few significant differences between them. The first one is the visibility rule. The

Table 1. The static methods in the Aspect class

Method	Description
void add(Object target, Object aspect, Class clazz)	assigns aspect to an anonymous field of target. clazz represents the type of the anonymous field.
void add(Object target, Collection aspects, Class clazz)	associates all the elements in aspects with target. clazz represents the class of the associated elements.
Object get(Object target, Class clazz)	obtains the value of an anonymous field of target. clazz represents the type of the anonymous field.
Collection getAll(Object target, Class clazz)	obtains the collection associated with target. clazz represents the type of the collection elements.
void remove(Object target, Object aspect, Class clazz)	unlinks aspect associated with target. clazz represents the type of the anonymous field.
void remove(Object target, Collection aspects, Class clazz)	unlinks all the elements in aspects associated with target. clazz represents the type of the collection elements.

advice body in GluonJ can access private members of the aspect target since it is glue code. On the other hand, the advice body in AspectJ cannot access except the members added by the intertype declarations. This is because the advice body in AspectJ belongs to the aspect implementation.

Another difference is how to specify which aspect implementation is associated with an aspect target. This section illustrates comparison between GluonJ and AspectJ with respect to this issue. Although GluonJ is similar to JBoss AOP and AspectWerkz rather than AspectJ, we compare GluonJ to AspectJ since the readers would be more familiar to AspectJ. In fact, AspectJ, JBoss AOP, and AspectWerkz are based on the same idea with respect to the association of aspect implementations. Note that, like GluonJ, JBoss AOP and AspectWerkz separate aspect bindings in XML from aspect implementation in Java. Although their aspect implementations are Java objects, they are implicitly constructed and associated as in AspectJ. On the other hand, an aspect implementation in GluonJ is explicitly constructed and associated.

4.1 Example

To illustrate that explicit association of aspect implementations in GluonJ enables a better expression of inter-component dependency than AspectJ, we present an implementation of simple caching mechanism in AspectJ and GluonJ. If a method always returns the same value when it is called with the same arguments, the returned value should be cached to improve the execution performance. Suppose that we would like to cache the result of the doExpensiveJob method in the following class:

```
public class MyTask {
  private int sessionId;
  public MyTask (int id) {
    sessionId = id;
  }
  public String doExpensiveJob(String s) {
    // the execution of this method takes a long time.
    // the result is computed from s and sessionId.
  }
}
```

Note that the returned value from doExpensiveJob depends only on the parameter s and the sessionId field. Thus we share cache memory among MyTask objects with the same session id.

We below see how GluonJ and AspectJ express the dependency between MyTask and the caching component. The goal is to implement the caching component to be independent of MyTask and naturally connect the two components by an aspect.

4.2 GluonJ

We first show the implementation in GluonJ (Figure 3). The following is the class for a caching component:

```
public class Cache {
  private HashMap cache = new HashMap();
  public Object getValue(JoinPoint thisJoinPoint, Object arg) {
    Object result = cache.get(arg);
    if (result == null) {
      try {
        result = thisJoinPoint.proceed();
        cache.put(arg, result);
      } catch (Throwable e) {}
    }
    return result;
  }
  // create a cache for each session.
  private static HashMap cacheMap = new HashMap();
  private static Cache factory(int sessionId) {
    Integer id = new Integer(sessionId);
    Cache c = (Cache)cacheMap.get(id);
    if (c == null) {
        c = new Cache();
        cacheMap.put(id, c);
    }
```

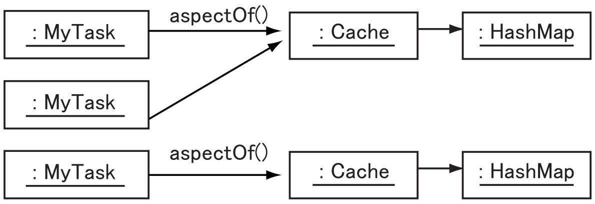


Fig. 3. The caching component in GluonJ

```
    return c;
  }
}
```

This component holds a hash table for caching the value returned from a method. factory is a factory method for constructing a Cache object for each session.

The Cache component is associated with a MyTask object. This association is described in the following aspect:

```
<aspect>
  <injection>
    Cache MyTask.aspect = Cache.factory(this.sessionId);
  </injection>
  <advice>
    <param><name>s</name> <type>String</type></param>
    <pointcut>
      execution(String MyTask.doExpensiveJob(..)) AND args(s)
    </pointcut>
    <around>
      return (String)Cache.aspectOf(this)
                          .getValue(thisJoinPoint, s);
    </around>
  </advice>
</aspect>
```

This aspect adds an anonymous field to MyTask. The value of this field is a Cache object for the session that the MyTask object belongs to. Then, if the doExpensiveJob method is executed, this aspect calls the getValue method on the associated Cache object.

Note that a Cache object is explicitly constructed in the aspect by calling a factory method. It is thereby associated with multiple MyTask objects belonging to the same session. The resulting object graph in Figure 3 naturally represents that the caching concern is per-session cache.

4.3 AspectJ (Using Intertype Declaration)

The caching mechanism can be also implemented in AspectJ. However, since AspectJ does not allow us to associate an aspect instance with a group of MyTask objects belonging to the same session, we must implement the per-session cache with a little bit complex programming. This is an example of the inflexibility for the implicit association of aspect instances in AspectJ. The following is an implementation using a singleton aspect and intertype field declaration (Figure 4):

```
privileged aspect CacheAspect {
  private HashMap MyTask.cache;    // intertype declaration

  after(MyTask t): execution(MyTask.new(..)) && this(t) {
    t.cache = factory(t.sessionId);
  }

  String around(MyTask t, String s): this(t) && args(s)
              && execution(String MyTask.doExpensiveJob(..)) {
```

```
    String result = (String)t.cache.get(s);
    if (result == null) {
      result = proceed(t, s);
      t.cache.put(s, result);
    }
    return result;
  }

  // create a cache for each session.
  private static HashMap cacheMap = new HashMap();
  private static HashMap factory(int sessionId) {
    Integer id = new Integer(sessionId);
    HashMap map = (HashMap)cacheMap.get(id);
    if (map == null) {
      map = new HashMap();
      cacheMap.put(id, map);
    }
    return map;
  }
}
```

Although the CacheAspect looks similar to the implementation in GluonJ, the resulting object-graph is different. It is far from the natural design. A single caching component, which is an instance of CacheAspect, manages the hash tables for all the sessions while each caching component in GluonJ manages a hash table for one session. Since there is only one caching component in AspectJ, a hash table for each MyTask object is stored in the cache field of the MyTask object. cache is the field added by intertype declaration. Hence the implementation of the caching concern is not only encapsulated within CacheAspect but also cutting across MyTask. Since AspectJ is a powerful aspect-oriented language, the implementation is not cutting across multiple components at the source-code level; it is cleanly modularized into CacheAspect. However, at the design level, the implementation of the caching concern involves MyTask. The developer must be aware that a hash table is contained in not CacheAspect but MyTask.

Another problem is that the caching concern is not really separated from other components since the dependency description (*i.e.* pointcut and advice) is

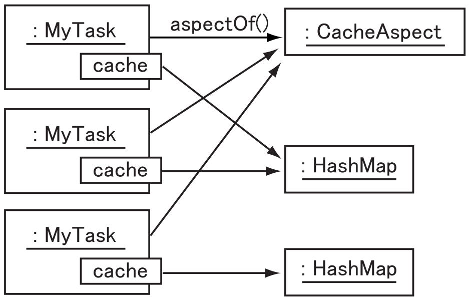


Fig. 4. The caching aspect using intertype declaration

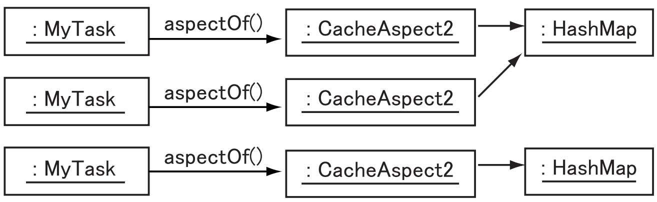


Fig. 5. The caching aspect using perthis

contained in the caching component. The caching component depends on MyTask since the class name MyTask is embedded in the intertype declaration in CacheAspect. If we reuse CacheAspect with another class other than MyTask, we must modify the definition of CacheAspect so that the cache field is added to that class. Although AspectJ provides abstract pointcut for parameterizing a class name occurring in a pointcut definition, it does not provide such a parameterization mechanism for intertype declarations.

Finally, since this aspect must access the sessionId field, which is private, it is declared as being privileged. A privileged aspect is not subject to the access control mechanism of Java. Thus, this implementation violates the encapsulation principle.

4.4 AspectJ (Using perthis)

The caching concern can be implemented with a perthis aspect (Figure 5). In the following implementation, an instance of CacheAspect2 is constructed for each MyTask object. This policy of aspect instantiation is specified by the perthis modifier. See the following program:

```
privileged aspect CacheAspect2 perthis(execution(* MyTask.*(..)) {
  private HashMap cache;    // aspect member

  after(MyTask t) : execution(MyTask.new(..)) && this(t) {
    cache = factory(t.sessionId);
  }

  String around(String s)
    : execution(String MyTask.doExpensiveJob(..)) && args(s) {
    String result = (String)cache.get(s);
    if (result == null) {
      result = proceed(s);
      cache.put(s, result);
    }
    return result;
  }

  // create a cache for each session.
  //      :
  // (the same as the factory method in CacheAspect)
}
```

Note that the hash table is stored in the cache field of the aspect instance. This aspect does not include intertype declaration. The cache field is a member of this aspect itself.

This implementation is simpler than the previous one since an instance of CacheAspect2 manages only one hash table stored in a field of that instance. CacheAspect2 does not have to access a field in MyTask. However, this implementation produces redundant aspect instances. The role of each aspect instance is merely a simple bridge between a MyTask object and a hash table. It has nothing elaborate. This is not appropriate from the viewpoint of either program design or efficiency.

Note that, in this implementation, both the caching component and the dependency description (with pointcuts and advice) are also tangled in CacheAspect2. However, separating the dependency description from the program of the caching component is not difficult if abstract pointcuts are used. We can define an aspect only for the caching mechanism and then define another aspect that extends the former aspect and implements the abstract pointcut for describing the dependency. The perthis modifier must be defined in the latter aspect.

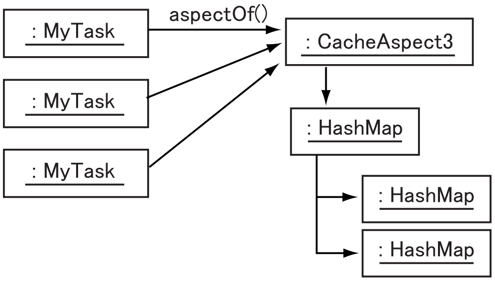


Fig. 6. The caching aspect using a hash table

4.5 AspectJ (Using a Hash Table)

The implementation we finally show uses a singleton aspect but it does not use an intertype field declaration or an aspect member field. In this implementation, either MyTask or CacheAspect3 do not include the cache field. The hash table is obtained from the factory method when the around advice is executed (Figure 6):

```
privileged aspect CacheAspect3 {
  String around(MyTask t, String s): this(t) && args(s)
               && execution(String MyTask.doExpensiveJob(..)) {
    HashMap cache = factory(t.sessionId); // obtain from factory()
    String result = (String)cache.get(s);
    if (result == null) {
      result = proceed(t, s);
      cache.put(s, result);
    }
    return result;
  }
  // create a cache for each session.
  //       :
  // (the same as the factory method in CacheAspect)
}
```

This would be the best implementation among the three AspectJ-based ones. The caching aspect is separated and independent of MyTask. No redundant aspect instance is produced. However, it is never highly efficient to call the factory method whenever the doExpensiveJob method is executed. Furthermore, this centralized design of caching mechanism is implementation-oriented. It would not be the design easily derived after the modeling phase. The easily derived design would be something like Figure 3 achieved by GluonJ. Figure 6 shown here would be the design that we could obtain by modifying that easily derived design to be suitable for implementation in a particular language.

Note that, in the implementation shown above, the dependency description (with pointcuts and advice) is also tangled with the caching component. However, separating the dependency description from the program of the caching component is possible by using abstract pointcuts.

5 Related Work

There are a number of aspect-oriented languages and frameworks that separate aspect binding and aspect implementation. Like GluonJ, JBoss AOP [9] and AspectWerkz [1] uses XML for describing aspect binding while Aspectual Components [13], Caesar [14] and JAsCo [19] uses extended language constructs. JAC [16] uses a programming framework in regular Java. Even AspectJ provides abstract aspects for this separation [8]. However, these systems allow only implicit association of an aspect implementation and hence they have a problem discussed in this paper. An aspect implementation is automatically constructed and implicitly associated with the aspect target in the specified scheme such as issingleton and perthis of AspectJ. Although JBoss AOP provides customization interface in Java for extending the behavior of perthis, it complicates the programming model.

The dynamic weaving mechanism of Caesar [14] allows associating an aspect implementation at runtime when the developers specify. It provides better flexibility but an aspect implementation is still automatically constructed and implicitly associated with the aspect target.

Association aspect [17] allows implementing a crosscutting concern by an explicitly constructed instance of an aspect. It is an extension to AspectJ and it is a language construct focusing on associating an aspect instance to a tuple of objects. GluonJ can be regarded as a framework generalizing the idea of association aspect and applying it to dependency reduction among components.

The implicit association of an aspect implementation (and an aspect instance in AspectJ) might be the ghost of the metaobject protocol [11], which is one of the origins of aspect-orientated programming. Although this design is not a problem if an aspect crosscuts only a single other concern, it should be revised to fully bring out the power of aspect orientation. Otherwise, advantages of aspect-oriented programming might be small against metaobject protocols.

AspectJ2EE [4] is an aspect-oriented programming system for J2EE. It restricts an aspect implementation to being associated with only a single aspect target. Therefore, it has the problem discussed in this paper.

Alice [5], JBoss AOP [9], and AspectWerkz [1] allow pointcuts that capture Java 5 annotations. This feature can be used for performing dependency injection on the fields annotated with `@inject`. Although this provides better syntax support, the developers must still define an aspect like `DependencyInjection` shown in Section 2.2.

The branch mechanism of Fred [15] provides basic functionality of aspect-oriented programming. It is similar to GluonJ since both of them provide only a dispatching mechanism based on pointcut and advice but they do not allow instantiation of aspects unlike AspectJ. However, Fred is a very simple Scheme-based language and it provides only a limited mechanism for dealing with dependency among components.

6 Conclusion

Reducing inter-component dependency is the goal of dependency injection but aspect-oriented programming can give a better solution to this goal. However, existing aspect-oriented programming systems have a problem. They can express only limited kinds of dependency relation since they implicitly associate an aspect implementation with an aspect target. The developers cannot fully control this relation.

To address this problem, this paper proposed *GluonJ*, which is our aspect-oriented framework for Java. A unique feature of GluonJ is that an aspect implementation is explicitly associated with aspect targets. An aspect in GluonJ consists of pointcuts and glue code written in Java. This glue code explicitly constructs an aspect implementation and associates it with appropriate aspect targets. The aspect implementation in GluonJ is a regular Java object.

We have implemented a prototype of GluonJ as a bytecode translator built on top of Javassist [2]. It supports most pointcut designators of AspectJ except `cflow`, which will be implemented in near future.

References

1. Boner, J., Vasseur, A.: AspectWerkz 1.0. `http://aspectwerkz.codehaus.org/` (2002)
2. Chiba, S.: Load-time structural reflection in Java. In: ECOOP 2000. LNCS 1850, Springer-Verlag (2000) 313–336
3. Chiba, S., Masuda, T.: Designing an extensible distributed language with a meta-level architecture. In: Proc. of the 7th European Conference on Object-Oriented Programming. LNCS 707, Springer-Verlag (1993) 482–501

4. Cohen, T., Gil, J.Y.: AspectJ2EE = AOP + J2EE : Towards an aspect based, programmable and extensible middleware framework. In: Proceedings of the European Conference on Object-Oriented Programming. Number 3086 in LNCS (2004) 219–243
5. Eichberg, M., Mezini, M.: Alice: Modularization of middleware using aspect-oriented programming. In: Software Engineering and Middleware (SEM) 2004. (2004)
6. Fowler, M.: Inversion of control containers and the dependency injection pattern. `http://www.martinfowler.com/articles/injection.html` (2004)
7. Golm, M., Kleinöder, J.: Jumping to the meta level, behavioral reflection can be fast and flexible. In: Proc. of Reflection '99. LNCS 1616, Springer (1999) 22–39
8. Hannemann, J., Kiczales, G.: Design pattern implementation in java and aspectj. In: Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications. (2002) 161–173
9. JBoss Inc.: JBoss AOP 1.0.0 final. `http://www.jboss.org/` (2004)
10. Johnson, R., Hoeller, J.: Expert One-on-One J2EE Development without EJB. Wrox (2004)
11. Kiczales, G., des Rivières, J., Bobrow, D.G.: The Art of the Metaobject Protocol. The MIT Press (1991)
12. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: ECOOP 2001 – Object-Oriented Programming. LNCS 2072, Springer (2001) 327–353
13. Lieberherr, K., Lorenz, D., Mezini, M.: Programming with Aspectual Components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA (1999)
14. Mezini, M., Ostermann, K.: Conquering aspects with caesar. In: Proc. of Int'l Conf. on Aspect-Oriented Software Development (AOSD'03), ACM Press (2003) 90–99
15. Orleans, D.: Incremental programming with extensible decisions. In: AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development, ACM Press (2002) 56–64
16. Pawlak, R., Seinturier, L., Duchien, L., Florin, G.: Jac: A flexible solution for aspect-oriented programming in java. In: Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001). LNCS 2192, Springer (2001) 1–24
17. Sakurai, K., Masuhara, H., Ubayashi, N., Matsuura, S., Kimoya, S.: Association aspects. In: Aspect-Oriented Software Development. (2004) 16–25
18. Smith, B.C.: Reflection and semantics in Lisp. In: Proc. of ACM Symp. on Principles of Programming Languages. (1984) 23–35
19. Suvée, D., Vanderperren, W., Jonckers, V.: Jasco: An aspect-oriented approach tailored for component based software development. In: Proc. of Int'l Conf. on Aspect-Oriented Software Development (AOSD'03), ACM Press (2003) 21–29
20. Welch, I., Stroud, R.: From dalang to kava — the evolution of a reflective java extension. In: Proc. of Reflection '99. LNCS 1616, Springer (1999) 2–21

Open Modules: Modular Reasoning About Advice

Jonathan Aldrich

Carnegie Mellon University, Pittsburgh, PA 15213, USA
jonathan.aldrich@cs.cmu.edu

Abstract. Advice is a mechanism used by advanced object-oriented and aspect-oriented programming languages to augment the behavior of methods in a program. Advice can help to make programs more modular by separating crosscutting concerns more effectively, but it also challenges existing ideas about modularity and separate development.

We study this challenge using a new, simple formal model for advice as it appears in languages like AspectJ. We then add a module system designed to leave program functionality as open to extension through advice as possible, while still enabling separate reasoning about the code within a module. Our system, Open Modules, can either be used directly to facilitate separate, component-based development, or can be viewed as a model of the features that certain AOP IDEs provide. We define a formal system for reasoning about the observational equivalence of programs under advice, which can be used to show that clients are unaffected by semantics-preserving changes to a module's implementation. Our model yields insights into the nature of modularity in the presence of advice, provides a mechanism for enforceable contracts between component providers and clients in this setting, and suggests improvements to current AOP IDEs.

1 Modularity and Advice

The Common Lisp Object System introduced a construct called *advice*, which allows a developer to externally augment the behavior of a method [2]. Advice comes in at least three flavors: *before* advice is run before the execution of a method body, *around* advice wraps a method body, and *after* advice runs after the method body. In general, advice can view or change the parameters or result of a method, or even control whether the method body is executed, allowing a rich set of adaptations to be implemented through this mechanism.

This paper examines advice in the context of Aspect-Oriented Programming (AOP), the most widely-used application of advice today [12]. The goal of AOP is to modularize concerns that crosscut the primary decomposition of a software system. AOP systems allow developers to modularize these *crosscutting concerns* within a single, locally-defined module, using an advice mechanism to allow definitions in that module to affect methods defined elsewhere in the system.

Although AOP in general and advice in particular provide many benefits to reasoning about concerns that are scattered and tangled throughout the code in conventional systems, questions have been raised about the ability to reason about and evolve code that is subject to advice. Advice appears to make reasoning about the effect of calling a method more challenging, for example, because it can intercept that call and change its semantics. In turn, this makes evolving AOP systems without tool support error-prone, because seemingly innocuous changes to base code could break the functionality of an aspect that advises that code.

For example, an important issue in separate development is ensuring that improvements and bug-fixes to a third-party component can be integrated into an application without breaking the application. Unfortunately, however, because the developer of a component does not know how is deployed, any change she makes could potentially break fragile pointcuts in a client [13] Although we could solve the problem by prohibiting all advice to third-party components, we would prefer a compromise that answers the research question:

1. *How can developers specify an interface for a library or component that permits as many uses of advice as possible, while still allowing the component to be changed in meaningful ways without affecting clients?*

Another important issue is protecting the internal invariants of component implementations in the presence of advice. For example, consider the Java standard library, which is carefully designed to provide a ”sandbox” security model for running untrusted code. In general, it is unsafe for a user to load any code that advises the standard library, because the advice could be used by an attacker to bypass the sandbox.

One possible solution to ensuring the security of the Java library is to prohibit all advice to the standard library. Unfortunately, this rule would prohibit many useful applications of advice. A better solution to the problem would allow as much advice as possible, while still preserving the internal invariants of the Java standard library. We thus use our formal model of modularity to address a second research question:

2. *How can developers specify an interface for a library or component that permits as many uses of advice as possible, while still ensuring correctness properties of the component implementation?*

The research questions above imply a solution that prohibits certain uses of advice in the case of separate development. However, in practice, some applications may only be able to reuse a library or component if they can get around these prohibitions. We think this should be an option for developers, but it should be a conscious choice: a developer should know when she is writing an aspect that may break when a new version of a component is released, and when she is writing an aspect that is resilient to new releases. Similarly, a user uploading code should know whether that code only includes advice that is guaranteed not to violate the Java sandbox, or whether the code contains advice that might violate the sandbox, and therefore ought to be signed by a trusted principal. Our research is aimed at providing developers and users with these informed choices.

1.1 Outline and Contributions

The next section of the paper describes *Open Modules*, a novel module system for advice that provides informal answers to the research questions. Open Modules is the first module system that supports many beneficial uses of advice, while still ensuring that security and other properties of a component implementation are maintained, and verifying that advice to a component will not be affected by behavior-preserving changes to the component's implementation.

We would like to make these answers precise, and to do so, Section 3 proposes `TinyAspect`, a novel formal model of AOP languages that captures a few core advice constructs while omitting complicating details. Section 4 extends `TinyAspect` with Open Modules, precisely defining the semantics and typing rules of the system.

Section 5 describes a formal system for reasoning about the observational equivalence of modules in the presence of advice. This is the first result to show that complete behavioral equivalence reasoning can be done on a module-by-module basis in a system with advice, providing a precise answer to research question 2. It also precisely defines what changes can be made to a component without affecting clients, answering research questions 1.

Section 6 discusses lessons learned from our formal model. Section 7 describes related work, and Section 8 concludes.

2 Open Modules

We propose Open Modules, a new module system for languages with advice that is intended to be *open* to extension with advice but *modular* in that the implementation details of a module are hidden. The goals of openness and modularity are in tension (at least in the case of separate development), and so we try to achieve a compromise between them.

In AOP systems, advice is used as a way to reach across module boundaries to capture crosscutting concerns. We propose to adopt the same advice constructs, but limit them so that they respect module boundaries. In order to capture concerns that crosscut the boundary of a module, we use AOP's *pointcut* abstraction to represent abstract sets of events that external modules may be interested in advising. As suggested by Gudmundson and Kiczales [9], exported pointcuts form a contract between a module and its client aspects, allowing the module to be evolved independently of its clients so long as the contract is preserved.

Figure 1 shows a conceptual view of Open Modules. Like ordinary module systems, open modules export a list of data structures and functions such as `moveBy` and `animate`. In addition, however, open modules can export pointcuts denoting internal semantic events. For example, the `moves` pointcut in Figure 1

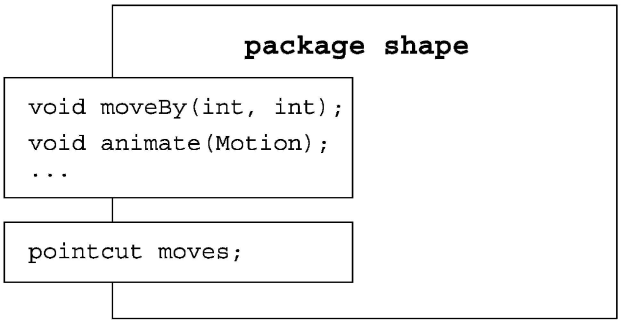


Fig. 1. A conceptual view of Open Modules. The `shape` module exports two functions and a pointcut. Clients can place advice on external calls to the exported functions, or on the exported pointcut, but not on calls that are internal to the module

is triggered whenever a shape moves. Since a shape could move multiple times during execution of the `animate` function, clients interested in fine-grained motion information would want to use this pointcut rather than just placing advice on calls to `animate`.

By exporting a pointcut, the module's maintainer is making a promise to maintain the semantics of that pointcut as the module's implementation evolves, just as the maintainer must maintain the semantics of the module's exported functions.

Open Modules are "open" in two respects. First, their interfaces are open to advice; all calls to interface functions from outside the module can be advised by clients. Second, clients can advise exported pointcuts.

On the other hand, open modules encapsulate the internal implementation details of a module. As usual with module systems, functions that are not exported in the module's public interface cannot be called from outside the module. In addition, in the case of separate development, calls between functions within the module cannot be advised from the outside—even if the called function is in the public interface of the module. For example, a client could place advice on external calls to `moveBy`, but not calls to `moveBy` from another function within the `shape` module.

In concurrent work, Kiczales and Mezini propose the notion of Aspect-Aware Interfaces, which are ordinary functional interfaces augmented with information about the advice that applies to a module. They point out that in a local development setting, analysis tools such as the AspectJ plugin for Eclipse (AJDT) [4] can compute aspect-aware interfaces automatically given whole-program information. Their work shows that in the case of local development and tool support, the benefits of Open Modules can be attained with no restrictions on the use of aspects. Instead, whenever an aspect that depends on internal calls is defined, the tools simply add a new pointcut to the module's aspect-aware interface, so that the new aspect conforms to the rules of Open Modules.

We now provide a canonical definition for Open Modules, which can be used to distinguish our contribution from previous work:

Definition [Open Modules]: *Open Modules describes a module system that:*

- *allows external advice to interactions between a module and the outside world (including external calls to functions in the interface of a module)*
- *allows external advice to pointcuts in the interface of a module*
- *does not allow external modules to directly advise internal events within the module, such as calls from within a module to other functions within the module (including calls to exported functions).*

3 Formally Modeling Advice

In order to reason formally and precisely about modularity, we need a formal model of advice. The most attractive models for this purpose are based on small-step operational semantics, which provide a very simple and direct formalization and are amenable to standard syntactic proof techniques.

Jagadeesan et al. have proposed an operational semantics for the core of AspectJ, incorporating several different kinds of pointcuts and advice in an object-oriented setting [10]. Their model is very rich and is thus ideal for specifying the semantics of a full language like AspectJ [11]. However, we would like to define and prove the soundness of a strong equivalence reasoning framework for the language, and doing so would be prohibitively difficult in such a complex model.

Walker et al. propose a much simpler formal model incorporating just the lambda calculus, advice, and labeled hooks that describe where advice may apply [19]. As a foundational calculus, their model is ideal for studying compilation strategies for AOP languages. However, because their model is low-level, it lacks some essential characteristics of advice in AOP, including the obliviousness property since advice applies to explicit labels [8]. The low-level nature of their language also means that certain properties of source-level languages like AspectJ—including the modularity properties we study—do not hold in their calculus. Thus, previous small-step operational models of aspects are inappropriate for our purposes.

Names	$n ::= x$
Expressions	$e ::= n \mid \texttt{fn}\ x{:}\tau\ \texttt{=>}\ e \mid e_1\ e_2 \mid \texttt{()}$
Declarations	$d ::= \bullet \mid \texttt{val}\ x = e\ \ d \mid \texttt{pointcut}\ x = p\ \ d \mid \texttt{around}\ p(x{:}\tau) = e\ \ d$
Pointcuts	$p ::= n \mid \texttt{call}(n)$
General exp.	$E ::= e \mid d \mid p$
Types	$\tau ::= \texttt{unit} \mid \tau_1 \rightarrow \tau_2$
Decl. Types	$\beta ::= \bullet \mid x{:}\tau, \beta \mid x{:}\pi, \beta$
Pcut. types	$\pi ::= \texttt{pc}(\tau_1 \rightarrow \tau_2)$
General types	$T ::= \tau \mid \beta \mid \pi$

Fig. 2. `TinyAspect` Source Syntax

3.1 TinyAspect

We have developed a new functional core language for aspect-oriented programming called `TinyAspect`. The `TinyAspect` language is intentionally small, making it feasible to rigorously prove strong properties such as the soundness of logical equivalence in Section 5. Although `TinyAspect` leaves out many features of full languages, we directly model advice constructs similar to those in AspectJ. Thus our model retains the declarative nature and oblivious properties of advice in existing AOP languages, helping to ensure that techniques developed in our model can be extended to full languages.

Because our paper is focused on studying modular reasoning for advice, we omit many of the powerful pointcut constructs of AOP languages like AspectJ. We do include simple pointcuts representing calls to a particular function in order to show how pointcuts in the interface of a module can contribute to separate reasoning in the presence of advice. Our system can easily be extended to other forms of static pointcuts, but an extension to a dynamic pointcut language with constructs like `cflow` [11] is beyond the scope of this work.

Figure 2 shows the syntax of `TinyAspect`. Our syntax is modeled after ML [18]. Names in `TinyAspect` are simple identifiers. Expressions include the monomorphic lambda calculus—names, functions, and function application. To this core, we add a primitive unit expression, so that we have a base case for types. We could add primitive booleans and integers in a completely standard way, and constructs like let can be encoded using lambdas. Since these constructs are orthogonal to aspects, we omit them for simplicity's sake.

In most aspect-oriented programming languages, including AspectJ, the pointcut and advice constructs are second-class and declarative. So as to be an accurate source-level model, a `TinyAspect` program is made up of a sequence of declarations. Each declaration defines a scope that includes the following declarations. A declaration is either the empty declaration, or a value binding, a pointcut binding, or advice. The `val` declaration gives a static name to a value so that it may be used or advised in other declarations.

The `pointcut` declaration names a pointcut in the program text. A pointcut of the form `call(`n`)` refers to any call to the function declaration n, while a pointcut of the form n is just an alias for a previous pointcut declaration n. The `around` declaration names some pointcut p describing calls to some function, binds the variable x to the argument of the function, and specifies that the advice e should be run in place of the original function. Inside the body of the advice e, the special variable `proceed` is bound to the original value of the function, so that e can choose to invoke the original function if desired.

`TinyAspect` types τ include the `unit` type and function types of the form $\tau_1 \rightarrow \tau_2$. We syntactically distinguish pointcut types π and declaration types β in order to enforce the second-class nature of these constructs (e.g., they cannot be computed by functions, nor can they be used to simulate fully general references).

3.2 Fibonacci Caching Example

We illustrate the language by writing the Fibonacci function in it, and writing a simple aspect that caches calls to the function to increase performance. While this is not a compelling example of aspects, it is standard in the literature and simple enough for an introduction to the language.

Figure 3 shows the `TinyAspect` code for the Fibonacci function. Integers, booleans, and if statements have been added to illustrate the example.

`TinyAspect` does not include a fixpoint operator for defining recursion, but advice can express the same thing. In the `fib` function above, we define the base case as an ordinary function definition, returning 1. We then place `around` advice that intercepts calls to `fib` and handles the recursive cases. The body of the advice checks to see if the argument is greater than 2; if so, it returns the sum of `fib(x-1)` and `fib(x-2)`. These recursive calls are intercepted by the advice, rather than the original function, allowing recursion to work properly. In the case when the argument is less than 3, the advice invokes `proceed` with the original number `x`. Within the scope of an advice declaration, the special variable `proceed` refers to the advised definition of the function. Thus, the call to `proceed` is forwarded to the original definition of `fib`, which returns 1.

In the lower half of the figure is an aspect that caches calls to `fib`, thereby allowing the normally exponential function to run in linear time. We assume there is a cache data structure and three functions for checking if a result is in the cache for a given value, looking up an argument in the cache, and storing a new argument-result pair in the cache.

So that we can make the caching code more reusable, we declare a `cacheFunction` pointcut that names the function calls to be cached—in this case, all calls to `fib`. Then we declare `around` advice on the `cacheFunction` pointcut which checks to see if the argument `x` is in the cache. If it is, the advice gets the result from the cache and returns it. If the value is not in the cache, the advice calls `proceed` to calculate the result of the call to `fib`, stores the result in the cache, and then returns the result.

```
val fib = fn x:int => 1
around call(fib) (x:int) =
    if (x > 2)
        then fib(x-1) + fib(x-2)
        else proceed x

(* advice to cache calls to fib *)
val inCache = fn ...
val lookupCache = fn ...
val updateCache = fn ...

pointcut cacheFunction = call(fib)
around cacheFunction(x:int) =
    if (inCache x)
        then lookupCache x
        else let v = proceed x
            in updateCache x v; v
```

Fig. 3. The Fibonacci function written in `TinyAspect`, along with an aspect that caches calls to `fib`

In the semantics of `TinyAspect`, the last advice to be declared on a declaration is invoked first. Thus, if a client calls `fib`, the caching advice will be invoked first. If the caching advice calls `proceed`, then the first advice (which recursively defines `fib`) will be invoked. If that advice in turn calls `proceed`, the original function definition will be invoked. However, if the advice makes a recursive call to `fib`, the call will be intercepted by the caching advice. Thus, the cache works exactly as we would expect—it is invoked on all recursive calls to `fib`, and thus it is able to effectively avoid the exponential cost of executing `fib` in the naïve way.

3.3 Operational Semantics

We define the semantics of `TinyAspect` more precisely as a set of small-step reduction rules. These rules translate a series of source-level declarations into the values shown in Figure 4.

Expression-level values include the unit value and functions. In `TinyAspect`, advice applies to declarations, *not* to functions. This is crucial for the modular reasoning result described later, as declarations can be hidden behind a module interface but first-class functions cannot. We therefore need to keep track of declaration usage in the program text, and so a reference to a declaration is represented by a label ℓ. In the operational semantics, below, an auxiliary environment keeps track of the advice that has been applied to each declaration.

A pointcut value can only take one form: calls to a particular declaration ℓ. In our formal system we model execution of declarations by replacing source-level declarations with "declaration values," which we distinguish by using the $\equiv$ symbol for binding.

Figure 4 also shows the contexts in which reduction may occur. Call-by-value reduction proceeds first on the left-hand side of an application, then on the right-hand side. Reduction occurs within a value declaration before proceeding to the following declarations. Pointcut declarations are atomic, and so they only define an evaluation context for the declarations that follow.

Figure 5 describes the operational semantics of `TinyAspect`. A machine state is a pair (η, e) of an advice environment η (mapping labels to values) and an expression e. Advice environments are similar to stores, but are used to keep track of a mapping from declaration labels to declaration values, and are modified by advice declarations. We use the $\eta[\ell]$ notation in order to look up the value of a label in η, and we denote the functional update of an environment as

$$\begin{array}{llcl}
\text{Expression values} & v & ::= & \texttt{()} \mid \texttt{fn}\ x{:}\tau\ \texttt{=>}\ e \mid \ell \\
\text{Declaration values} & d_v & ::= & \bullet \mid \texttt{val}\ x \equiv \ell\ \ d_v \mid \texttt{pointcut}\ x \equiv \texttt{call}(\ell)\ \ d_v \\
\text{Evaluation contexts} & C & ::= & \Box\ e_2 \mid v_1\ \Box \mid \texttt{val}\ x = \Box\ \ d \\
 & & \mid & \mathit{bind}\ x \equiv V\ \ \Box \mid \texttt{pointcut}\ x = \Box\ \ d \\
\text{General values} & V & ::= & v \mid d_v \mid \texttt{call}(\ell)
\end{array}$$

Fig. 4. `TinyAspect` Values and Contexts

$$\frac{}{(\eta,\ (\texttt{fn}\ x{:}\tau\ \texttt{=>}\ e)\ v) \mapsto (\eta,\ \{v/x\}e)}\ \textit{r-app} \qquad \frac{\eta[\ell] = v_1}{(\eta,\ \ell\ v_2) \mapsto (\eta,\ v_1\ v_2)}\ \textit{r-lookup}$$

$$\frac{\ell \notin \textit{domain}(\eta) \quad \eta' = [\ell \mapsto v]\ \eta}{(\eta,\ \texttt{val}\ x = v\ \ d) \mapsto (\eta',\ \texttt{val}\ x \equiv \ell\ \ \{\ell/x\}d)}\ \textit{r-val}$$

$$\frac{}{\begin{array}{c}(\eta,\ \texttt{pointcut}\ x = \texttt{call}(\ell)\ \ d) \mapsto \\ (\eta,\ \texttt{pointcut}\ x \equiv \texttt{call}(\ell)\ \ \{\texttt{call}(\ell)/x\}d)\end{array}}\ \textit{r-pointcut}$$

$$\frac{\begin{array}{l}v' = (\texttt{fn}\ x{:}\tau\ \texttt{=>}\ \{\ell'/\texttt{proceed}\}e) \\ \ell' \notin \textit{domain}(\eta) \qquad \eta' = [\ell \mapsto v', \ell' \mapsto \eta[\ell]]\ \eta\end{array}}{(\eta,\ \texttt{around}\ \texttt{call}(\ell)(x{:}\tau) = e\ \ d) \mapsto (\eta',\ d)}\ \textit{r-around} \qquad \frac{(\eta,\ e) \mapsto (\eta',\ e')}{(\eta,\ C[e]) \mapsto \eta',\ C[e'])}\ \textit{r-ctx}$$

Fig. 5. `TinyAspect` Operational Semantics

$\eta' = [\ell \mapsto v]\ \eta$. The reduction judgment is of the form $(\eta, e) \mapsto (\eta', e')$, read, "In advice environment η, expression e reduces to expression e' with a new advice environment η'."

The rule for function application is standard, replacing the application with the body of the function and substituting the argument value v for the formal x. We normally treat labels ℓ as values, and there is no rule $\ell \mapsto \eta[\ell]$ because we want to avoid "looking them up" before they are advised. However, when we are in a position to invoke the function represented by a label, we use the rule *r-lookup* to look up the label's value in the current environment.

The next three rules reduce declarations to "declaration values." The `val` declaration binds the value to a fresh label and adds the binding to the current environment. It also substitutes the label for the variable x in the subsequent declaration(s) d. We leave the binding in the reduced expression both to make type preservation easier to prove, and also to make it easy to extend `TinyAspect` with a module system which will need to retain the bindings. The `pointcut` declaration simply substitutes the pointcut value for the variable x in subsequent declaration(s).

The `around` declaration looks up the advised declaration ℓ in the current environment. It places the old value for the binding in a fresh label ℓ', and then re-binds the original ℓ to the body of the advice. Inside the advice body, any references to the special variable `proceed` are replaced with ℓ', which refers to the original value of the advised declaration. Thus, all references to the original declaration will now be redirected to the advice, while the advice can still invoke the original function by calling `proceed`.

The last rule shows that reduction can proceed under any context as defined in Figure 4.

3.4 Typechecking

Figure 6 describes the typechecking rules for `TinyAspect`. Our typing judgment for expressions is of the form $\Gamma; \Sigma \vdash e : \tau$, read, "In variable context Γ and

$$\frac{x{:}\tau \in \Gamma}{\Gamma;\Sigma \vdash x : \tau}\ \textit{t-var} \qquad \frac{\Gamma;\Sigma \vdash n : \tau_1 \rightarrow \tau_2}{\Gamma;\Sigma \vdash \mathtt{call}(n) : \mathtt{pc}(\tau_1 \rightarrow \tau_2)}\ \textit{t-pctype}$$

$$\frac{\ell{:}\tau \in \Sigma}{\Gamma;\Sigma \vdash \ell : \tau}\ \textit{t-label} \qquad \frac{}{\Gamma;\Sigma \vdash \mathtt{()} : \mathtt{unit}}\ \textit{t-unit}$$

$$\frac{\Gamma, x{:}\tau_1;\Sigma \vdash e : \tau_2}{\Gamma;\Sigma \vdash \mathtt{fn}\ x{:}\tau_1\ \mathtt{=>}\ e : \tau_1 \rightarrow \tau_2}\ \textit{t-fn} \qquad \frac{\Gamma;\Sigma \vdash e_1 : \tau_2 \rightarrow \tau_1 \quad \Gamma;\Sigma \vdash e_2 : \tau_2}{\Gamma;\Sigma \vdash e_1\ e_2 : \tau_1}\ \textit{t-app}$$

$$\frac{}{\Gamma;\Sigma \vdash \bullet : \bullet}\ \textit{t-empty} \qquad \frac{\Gamma;\Sigma \vdash v : T \quad \Gamma;\Sigma \vdash d : \beta}{\Gamma;\Sigma \vdash \textit{bind}\ x \equiv v\ \ d : (x{:}T, \beta)}\ \textit{t-vdecl}$$

$$\frac{\Gamma;\Sigma \vdash e : \tau \quad \Gamma, x{:}\tau;\Sigma \vdash d : \beta}{\Gamma;\Sigma \vdash \mathtt{val}\ x = e\ \ d : (x{:}\tau, \beta)}\ \textit{t-val} \qquad \frac{\Gamma;\Sigma \vdash p : \pi \quad \Gamma, x{:}\pi;\Sigma \vdash d : \beta}{\Gamma;\Sigma \vdash \mathtt{pointcut}\ x = p\ \ d : (x{:}\pi, \beta)}\ \textit{t-pc}$$

$$\frac{\begin{array}{c}\Gamma;\Sigma \vdash p : \mathtt{pc}(\tau_1 \rightarrow \tau_2) \qquad \Gamma;\Sigma \vdash d : \beta \\ \Gamma, x{:}\tau_1, \mathtt{proceed}{:}\tau_1 \rightarrow \tau_2;\Sigma \vdash e : \tau_2\end{array}}{\Gamma;\Sigma \vdash \mathtt{around}\ p(x{:}\tau_1) = e\ \ d : \beta}\ \textit{t-around}$$

$$\frac{\forall \ell \in \textit{domain}(\Sigma)\ .\ \ \bullet;\Sigma \vdash \eta[\ell] : \Sigma[\ell])}{\Sigma \vdash \eta}\ \textit{t-env}$$

Fig. 6. `TinyAspect` Typechecking

declaration context Σ expression e has type τ." Here Γ maps variable names to types, while Σ maps labels to types (similar to a store type).

The rules for expressions are standard. We look up the types for variables and labels in Γ and Σ, respectively. Other standard rules give types to the `()` expression, as well as to functions and applications.

The interesting rules are those for declarations. We give declaration signatures β to declarations, where β is a sequence of variable to type bindings. The base case of an empty declaration has an empty signature. For `val` bindings, we ensure that the expression is well-typed at some type τ, and then typecheck subsequent declarations assuming that the bound variable has that type. Pointcuts are similar, but the rule ensures that the expression p is well-typed as a pointcut denoting calls to a function of type $\tau_1 \rightarrow \tau_2$. When a val or pointcut binding becomes a value, the typing rule is the same except that subsequent declarations cannot see the bound variable (as it has already been substituted in). The around advice rule checks that the declared type of x matches the argument type in the pointcut, and checks that the body is well-typed assuming proper types for the variables x and `proceed`.

Finally, the judgment $\Sigma \vdash \eta$ states that η is a well-formed environment with typing Σ whenever all the values in η have the types given in Σ. This judgment, used in the soundness theorem, is analogous to store typings in languages with references.

3.5 Type Soundness

We now state progress and preservation theorems for `TinyAspect`. The theorems quantify over both expressions and declarations using the metavariable E, and quantify over types and declaration signatures using the metavariable T. The progress property states that if an expression is well-typed, then either it is already a value or it will take a step to some new expression.

Theorem 1 (Progress). *If* $;\Sigma \vdash E : T$ *and* $\Sigma \vdash \eta$*, then either* E *is a value or there exists* η' *such that* $(\eta, E) \mapsto (\eta', E')$*.*

Proof. By induction on the derivation of $;\Sigma \vdash E : T$.

The type preservation property states that if an expression is well-typed and it reduces to another expression in a new environment, then the new expression and environment are also well-typed.

Theorem 2 (Type Preservation). *If* $;\Sigma \vdash E : T$*,* $\Sigma \vdash \eta$*, and* $(\eta, E) \mapsto (\eta', E')$*, then there exists some* $\Sigma' \supseteq \Sigma$ *such that* $;\Sigma' \vdash E' : T$ *and* $\Sigma' \vdash \eta'$*.*

Proof. By induction on the derivation of $(\eta, E) \mapsto (\eta', E')$. The proof relies on standard substitution and weakening lemmas.

Together, progress and type preservation imply type soundness. Soundness means that there is no way that a well-typed `TinyAspect` program can get stuck or "go wrong" because it gets into some bad state.

Our type soundness theorem is slightly stronger than the previous result of Walker et al., in that we guarantee both type safety and a lack of run time errors. Walker et al. model `around` advice using a lower-level exception construct, and so their soundness theorem includes the possibility that the program will terminate with an uncaught exception [19].

4 Formalizing Modules

We now extend `TinyAspect` with Open Modules, a module system that allows programmers to enforce an abstraction boundary between clients and the implementation of a module. Our module system is modeled closely after that of ML, providing a familiar concrete syntax and benefiting from the design of an already advanced module system. In a distributed development setting, our module system places restrictions on aspects in order to provide the strong reasoning guarantee in Section 5. In a local development setting, however, our "module interfaces" could be computed by tools, and place no true restrictions on developers.

Figure 7 shows the new syntax for modules. Names include both simple variables x and qualified names $m.x$, where m is a module expression. Declarations can include structure bindings, and types are extended with module signatures of the form `sig` β, where β is the list of variable to type bindings in the module signature.

Names	$n ::= \ldots \mid m.x$
Declarations	$d ::= \ldots \mid \texttt{structure}\ x = m\ \ d$
Modules	$m ::= n \mid \texttt{struct}\ d\ \texttt{end} \mid m\ \texttt{:>}\ \sigma \mid \texttt{functor}(x{:}\sigma)\ \texttt{=>}\ m \mid m_1\ m_2$
Decl. types	$\beta ::= \ldots \mid x{:}\sigma, \beta$
Module types	$\sigma ::= \texttt{sig}\ \beta\ \texttt{end} \mid \sigma_1 \rightarrow \sigma_2$
Module values	$m_v ::= \texttt{struct}\ d_v\ \texttt{end} \mid \texttt{functor}(x{:}\sigma)\ \texttt{=>}\ m$
Contexts	$C ::= \ldots \mid \texttt{structure}\ x = \square\ \ d \mid \texttt{struct}\ \square\ \texttt{end}$
	$\mid\ \ \square\ \texttt{:>}\ \sigma \mid \square\ m_2 \mid m_v\ \square$

Fig. 7. Module System Syntax, Values, and Contexts

```
structure Cache = functor(X : sig f : pc(int->int) end) =>
    struct
        around X.f(x:int) = (* same definition as before *)
    end

structure Math = struct
    val fib = fn x:int => 1
    around call(fib) (x:int) =
        if (x > 2)
            then fib(x-1) + fib(x-2)
            else proceed x

    structure cacheFib =
        Cache (struct pointcut f = call(fib) end)

end :> sig
    fib : int->int
end
```

Fig. 8. Fibonacci with Open Modules

First-order module expressions include a name, a `struct` with a list of declarations, and an expression m `:>` σ that seals a module with a signature, hiding elements not listed in the signature. The expression `functor` $x{:}\sigma$ `=>` m describes a functor that takes a module x with signature σ as an argument, and returns the module m which may depend on x. Functor application is written like function application, using the form $m_1\ m_2$.

4.1 Fibonacci Revisited

Figure 8 shows how a more reusable caching aspect could be defined using functors. The `Cache` functor accepts a module that has a single element `f` that is a pointcut of calls to some function with signature `int->int`. The `around` advice then advises the pointcut from the argument module `X`.

The `fib` function is now encapsulated inside the `Math` module. The module implements caching by instantiating the `Cache` module with a structure that binds the pointcut `f` to calls to `fib`. Finally, the `Math` module is sealed with a signature that exposes only the `fib` function to clients.

4.2 Sealing

Our module sealing operation has an effect both at the type system level and at the operational level. At the type level, it hides all members of a module that are not in the signature σ—in this respect, it is similar to sealing in ML's module system. However, sealing also has an operational effect, hiding internal calls within the module so that in a distributed development setting, clients cannot advise them unless the module explicitly exports the corresponding pointcut.

For example, in Figure 8, clients of the `Math` module would not be able to tell whether or not caching had been applied, even if they placed advice on `Math.fib`. Because `Math` has been sealed, external advice to `Math.fib` would only be invoked on external calls to the function, not on internal, recursive calls. This ensures that clients cannot be affected if the implementation of the module is changed, for example, by adding or removing caching.

4.3 Exposing Semantic Events with Pointcuts

Figure 9 shows how the shape example described above could be modeled in `TinyAspect`. Clients of the shape library cannot advise internal functions, because the module is sealed. To allow clients to observe internal but semantically important events like the motion of animated shapes, the module exposes these events in its signature as the `moves` pointcut. Clients can advise this pointcut without depending on the internals of the shape module. If the module's implementation is changed, the `moves` pointcut must also be updated so that client aspects are triggered in the same way.

Explicitly exposing internal events in an interface pointcut means a loss of some obliviousness in the distributed development case, since the author of the module must anticipate that clients might be interested in the event. On the other hand, we are still better off than in a non-AOP language, because the interface pointcut is defined in a way that does not affect the actual implementation of the module, as opposed to an invasive explicit callback, and because external calls to interface functions can still be obliviously advised.

Thus, sealing enforces the abstraction boundary between a module and its clients, allowing programmers to reason about and change them independently. However, our system still allows a module to export semantically important

```
structure shape = struct
    val createShape = fn ...
    val moveBy = fn ...
    val animate = fn ...
    ...
    pointcut moves = call(moveBy)
end :> sig
    createShape : Description -> Shape
    moveBy      : (Shape,Location) -> unit
    animate     : (Shape,Path) -> unit
    ...
    moves       : pc((Shape,Location)->unit)
end
```

Fig. 9. A shape library that exposes a position change pointcut

$$\frac{\mathit{bind}\ x \equiv v \in d_v}{(\eta,\ \texttt{struct}\ d_v\ \texttt{end}.x) \mapsto (\eta,\ v)}\ \textit{r-path} \qquad \frac{}{\begin{array}{l}(\eta,\ \texttt{structure}\ x = m_v\ \ d) \mapsto \\ (\eta,\ \texttt{structure}\ x \equiv m_v\ \ \{m_v/x\}d)\end{array}}\ \textit{r-struct}$$

$$\frac{}{\begin{array}{l}(\eta,\ (\texttt{functor}(x{:}\sigma)\ \texttt{=>}\ m_1)\ m_v) \\ \quad\mapsto (\eta,\ \{m_v/x\}m_1)\end{array}}\ \textit{r-fapp} \qquad \frac{\mathit{seal}(\eta, d_v, \beta) = (\eta', d_{seal})}{\begin{array}{l}(\eta,\ \texttt{struct}\ d_v\ \texttt{end :> sig}\ \beta\ \texttt{end}) \\ \quad\mapsto (\eta',\ \texttt{struct}\ d_{seal}\ \texttt{end})\end{array}}\ \textit{r-seal}$$

$$\frac{}{\mathit{seal}(\eta, \bullet, \bullet) = (\eta, \bullet)}\ \textit{s-empty} \qquad \frac{\mathit{seal}(\eta, d, \beta) = (\eta', d')}{\mathit{seal}(\eta, \mathit{bind}\ x \equiv v\ \ d, \beta) = (\eta', d')}\ \textit{s-omit}$$

$$\frac{\mathit{seal}(\eta, d, \beta) = (\eta', d') \quad \eta'' = [\ell \mapsto \ell']\ \eta' \quad \ell \notin \mathit{domain}(\eta')}{\mathit{seal}(\eta, \texttt{val}\ x \equiv \ell'\ \ d, (x{:}\tau, \beta)) = (\eta'', \texttt{val}\ x \equiv \ell\ \ d')}\ \textit{s-v}$$

$$\frac{\mathit{seal}(\eta, d, \beta) = (\eta', d')}{\mathit{seal}(\eta, \texttt{pointcut}\ x \equiv \texttt{call}(\ell)\ \ d, (x{:}\mathrm{pc}(\tau), \beta)) = (\eta', \texttt{pointcut}\ x \equiv \texttt{call}(\ell)\ \ d')}\ \textit{s-p}$$

$$\frac{\mathit{seal}(\eta, d_s, \beta_s) = (\eta', d_s') \quad \mathit{seal}(\eta', d, \beta) = (\eta'', d')}{\begin{array}{r}\mathit{seal}(\eta, \texttt{structure}\ x \equiv \texttt{struct}\ d_s\ \texttt{end}\ \ d, (x{:}\texttt{sig}\ \beta_s\ \texttt{end}, \beta)) \\ = (\eta'', \texttt{structure}\ x \equiv \texttt{struct}\ d_s'\ \texttt{end}\ \ d')\end{array}}\ \textit{s-s}$$

$$\frac{\mathit{seal}(\eta, d, \beta) = (\eta', d')}{\begin{array}{l}\mathit{seal}(\eta, \texttt{structure}\ x \equiv \texttt{functor}(y{:}\sigma_y)\ \texttt{=>}\ m\ \ d, (x{:}\sigma, \beta)) \\ \quad = (\eta', \texttt{structure}\ x \equiv \texttt{functor}(y{:}\sigma_y)\ \texttt{=>}\ m\ \ d')\end{array}}\ \textit{s-f}$$

Fig. 10. Module System Operational Semantics

internal events, allowing clients to extend or observe the module's behavior in a principled way.

4.4 Operational Semantics

Figure 10 shows the operational semantics for Open Modules. In the rules, module values m_v mean either a struct with declaration values d_v or a functor. The path lookup rule finds the selected binding within the declarations of the module. We assume that bound names are distinct in this rule; it is easy to ensure this by renaming variables appropriately. Because modules cannot be advised, there is no need to create labels for structure declarations; we can just substitute the structure value for the variable in subsequent declarations. The rule for functor application also uses substitution.

The rule for sealing uses an auxiliary judgment, *seal*, to generate a fresh set of labels for the bindings exposed in the signature. This fresh set of labels insures that clients can affect external calls to module functions by advising the new labels, but cannot advise calls that are internal to the sealed module.

At the bottom of the diagram are the rules defining the sealing operation. The operation accepts an old environment η, a list of declarations d, and the sealing

$$\frac{\Gamma; \Sigma \vdash m : \texttt{sig}\ \beta\ \texttt{end} \quad x{:}\tau \in \beta}{\Gamma; \Sigma \vdash m.x : \tau}\ \textit{t-name} \qquad \frac{\Gamma; \Sigma \vdash m : \sigma \quad \Gamma, x{:}\sigma; \Sigma \vdash d : \beta}{\Gamma; \Sigma \vdash \texttt{structure}\ x = m\ \ d : (x{:}\sigma, \beta)}\ \textit{t-str}$$

$$\frac{\Gamma; \Sigma \vdash d : \beta}{\Gamma; \Sigma \vdash \texttt{struct}\ d\ \texttt{end} : \texttt{sig}\ \beta\ \texttt{end}}\ \textit{t-struct} \qquad \frac{\Gamma; \Sigma \vdash m : \sigma_m \quad \sigma_m \texttt{ <: } \sigma}{\Gamma; \Sigma \vdash m \texttt{ :> } \sigma : \sigma}\ \textit{t-seal}$$

$$\frac{\Gamma, x{:}\sigma_1; \Sigma \vdash m : \sigma_2}{\Gamma; \Sigma \vdash \texttt{functor}(x{:}\sigma_1)\ \texttt{=>}\ m : \sigma_1 \rightarrow \sigma_2}\ \textit{t-ftor} \qquad \frac{\begin{array}{c}\Gamma; \Sigma \vdash m_1 : \sigma_1 \rightarrow \sigma \\ \Gamma; \Sigma \vdash m_2 : \sigma_2 \qquad \sigma_2 \texttt{ <: } \sigma_1\end{array}}{\Gamma; \Sigma \vdash m_1\ m_2 : \sigma}\ \textit{t-fapp}$$

Fig. 11. Open Modules Typechecking

$$\frac{}{\sigma \texttt{ <: } \sigma}\ \textit{sub-reflex} \qquad \frac{\sigma \texttt{ <: } \sigma' \quad \sigma' \texttt{ <: } \sigma''}{\sigma \texttt{ <: } \sigma''}\ \textit{sub-trans}$$

$$\frac{\beta \texttt{ <: } \beta'}{\texttt{sig}\ \beta\ \texttt{end}\ \texttt{ <: }\ \texttt{sig}\ \beta'\ \texttt{end}}\ \textit{sub-sig} \qquad \frac{\beta \texttt{ <: } \beta'}{x : T, \beta\ \texttt{ <: }\ \beta'}\ \textit{sub-omit}$$

$$\frac{\beta \texttt{ <: } \beta' \quad \sigma \texttt{ <: } \sigma'}{x : \sigma, \beta\ \texttt{ <: }\ x : \sigma', \beta'}\ \textit{sub-decl} \qquad \frac{\sigma_1' \texttt{ <: } \sigma_1 \quad \sigma_2 \texttt{ <: } \sigma_2'}{\sigma_1 \rightarrow \sigma_2\ \texttt{ <: }\ \sigma_1' \rightarrow \sigma_2'}\ \textit{sub-contra}$$

Fig. 12. Signature Subtyping

declaration signature β. The operation computes a new environment η' and new list of declarations d'. The rules are structured according to the first declaration in the list; each rule handles the first declaration and appeals recursively to the definition of sealing to handle the remaining declarations.

An empty list of declarations can be sealed with the empty signature, resulting in another empty list of declarations and an unchanged environment η. The second rule allows a declaration *bind* $x \equiv v$ (where *bind* represents one of `val`, `pointcut`, or `struct`) to be omitted from the signature, so that clients cannot see it at all. The rule for sealing a value declaration generates a fresh label ℓ, maps that to the old value of the variable binding in η, and returns a declaration mapping the variable to ℓ. Client advice to the new label ℓ will affect only external calls, since internal references still refer to the old label which clients cannot change. The rule for pointcuts passes the pointcut value through to clients unchanged, allowing clients to advise the label referred to in the pointcut. Finally, the rules for structure declarations recursively seal any internal struct declarations, but leave functors unchanged.

4.5 Typechecking

The typechecking rules, shown in Figure 11, are largely standard. Qualified names are typed based on the binding in the signature of the module m. Structure bindings are given a declaration signature based on the signature σ of the bound module. The rule for `struct` simply puts a `sig` wrapper around the declaration signature. The rules for sealing and functor application allow a module to be passed into a context where a supertype of its signature is expected.

Figure 12 shows the definition of signature subtyping. Subtyping is reflexive and transitive. Subtype signatures may have additional bindings, and the signatures of constituent bindings are covariant. Finally, the subtyping rule for functor types is contravariant.

4.6 Type Soundness

When extended with Open Modules, `TinyAspect` enjoys the same type soundness property that the base system has. The theorems and proofs are similar, and so we omit them.

5 Reasoning About Equivalence

The example programs in Section 4 are helpful for understanding the benefits of `TinyAspect`'s module system at an intuitive level. However, we would like to be able to point to a concrete property that enables separate reasoning about the clients and implementation of a module.

Asking whether the implementation of a module is correct, or whether changes can be made to the module without affecting clients, is asking about the equivalence between a module implementation and a specification or between two module implementations. For the purposes of this paper, we assume that a specification is given as a reference implementation, reducing both questions to comparing two implementations. This definition is limited, since many specifications are intended to leave some behavior up to the implementor, but we leave a more flexible definition to future work.

A natural definition of equivalence is called *observational equivalence* [17] or *contextual equivalence*, meaning that no client context can distinguish two different implementations of a component. A simple way to define contextual equivalence is to use program termination as the observable variable: two expressions in a program are contextually equivalent if, for all client contexts, the client will either terminate when linked to both implementations of a component, or will run forever when linked to both implementations. We formalize this as follows:

Definition [Contextual Equivalence]: *Two expressions* E_1 *and* E_2 *are contextually equivalent, written* $E_1 \equiv E_2$*, if and only if for all contexts* C *such that* $;\vdash C[E_1] : \tau$ *and* $;\vdash C[E_2] : \tau$ *we have* $(,E_1) \mapsto^* (\eta_1, V_1) \Longleftrightarrow (,E_2) \mapsto^* (\eta_2, V_2)$.

By definition, two contextually equivalent modules cannot be distinguished by any client.[1] Thus, contextual equivalence is adequate to answer whether a

[1] Note that since our formal system only models functional behavior, it cannot distinguish implementations with different performance characteristics.

$$\frac{(\bullet, E_1)\ \mathit{diverges} \qquad (\bullet, E_1)\ \mathit{diverges}}{E_1 \cong E_2}$$

$$\frac{\begin{array}{c}(\bullet, E_1) \mapsto^* (\eta_1, V_1) \qquad (\bullet, E_2) \mapsto^* (\eta_2, V_2) \qquad \Lambda, \Sigma \vdash (\eta_1, V_1) \simeq (\eta_2, V_2) : T \\ \Lambda = (\mathit{domain}(\eta_1) \cup \mathit{domain}(\eta_2)) - (\mathit{fl}(V_1) \cup \mathit{fl}(V_2)) \qquad \Lambda, \Sigma \vdash \eta_1 \simeq \eta_2\end{array}}{E_1 \cong E_2}$$

Fig. 13. Logical Equivalence for Program Text

change to a module might affect clients, or whether an optimized implementation of a module is semantically equivalent to a reference implementation.

5.1 Logical Equivalence

Although contextual equivalence intuitively captures the semantics of equivalence, it is not very useful for actually proving that two modules are equivalent, because it requires quantifying over all possible clients. Instead, we give a more useful set of *logical equivalence* rules that can be used to reason about a module in isolation from possible clients. We then prove that these rules are sound with respect to the more natural, but less useful, contextual equivalence semantics. Finally, we briefly outline how the logical equivalence rules can be used to prove that two different implementations of a module are observationally equivalent.

Figure 13 defines logical equivalence for `TinyAspect` expressions. If two expressions diverge, they are logically equivalent. Otherwise, two expressions are equivalent if, in the empty context, they both reduce to environment-value pairs that obey a value equivalence relation, defined below. Note that these equivalence rules apply only to closed expressions; this is not a significant limitation, as expressions with free variables can easily be re-written as functions or functors.

The equivalence relation for values is somewhat more complex, because whether two values are equivalent depends both on the environment bindings for free labels in the value, and on which labels clients can advise. For example, the `Math` module in Figure 8 is equivalent to the same module without caching if clients cannot advise the internal recursive calls to `fib`, but would not be equivalent if clients can advise these calls.

We define an value equivalence judgment of the form $\Lambda, \Sigma \vdash (\eta, V) \simeq (\eta', V')$: T. Here, Λ represents a set of hidden labels that a client cannot advise. Σ is the type of labels that the client can advise (i.e., those not in Λ). The equivalence judgment includes both values and their corresponding environments, because whether two values are equivalent may depend on how they each use their private labels in Λ. A similar judgment, $\Lambda, \Sigma \vdash \eta_1 \simeq \eta_2$, used in the second logical equivalence rule and defined in Figure 14, verifies that two environments map all labels not in Λ to logically equivalent values.

The second rule in Figure 13 sets Λ to be all of the labels in the two environments that are not free in the values being compared (the set of free labels in V is written $\mathit{fl}(V)$). For the `Math` module in Figure 8, only the label generated for the `fib` function as part of the module sealing operation is free in the module value; all other labels, including the one that captures advice on

$$\frac{}{\Lambda, \Sigma \vdash (\eta_1, \texttt{()}) \simeq (\eta_2, \texttt{()}) : \texttt{unit}} \qquad \frac{\ell \notin \Lambda}{\Lambda, \Sigma \vdash (\eta_1, \ell) \simeq (\eta_2, \ell) : \Sigma[\ell]}$$

$$\frac{\ell \in \Lambda \qquad \Lambda, \Sigma \vdash (\eta_1, \eta_1[\ell]) \simeq (\eta_2, v) : \tau}{\Lambda, \Sigma \vdash (\eta_1, \ell) \simeq (\eta_2, v) : \tau} \qquad \frac{\ell \in \Lambda \qquad \Lambda, \Sigma \vdash (\eta_1, v) \simeq (\eta_2, \eta_2[\ell]) : \tau}{\Lambda, \Sigma \vdash (\eta_1, v) \simeq (\eta_2, \ell) : \tau}$$

$$\frac{\ell \notin \Lambda \qquad \Lambda, (\Sigma, \ell{:}\tau') \vdash (\eta_1, (\texttt{fn}\ x{:}\tau'\ \texttt{=>}\ e_1)\ \ell) \cong (\eta_2, (\texttt{fn}\ x{:}\tau'\ \texttt{=>}\ e_2)\ \ell) : \tau}{\Lambda, \Sigma \vdash (\eta_1, \texttt{fn}\ x{:}\tau'\ \texttt{=>}\ e_1) \simeq (\eta_2, \texttt{fn}\ x{:}\tau'\ \texttt{=>}\ e_2) : \tau' \to \tau}$$

$$\frac{}{\Lambda, \Sigma \vdash (\eta, \bullet) \simeq (\eta', \bullet) : (\bullet)} \qquad \frac{\ell \notin \Lambda \qquad \Lambda, \Sigma \vdash (\eta, d_v) \simeq (\eta', d_v') : \beta}{\Lambda, \Sigma \vdash (\eta, \texttt{val}\ x \equiv \ell\ \ d_v) \simeq (\eta', \texttt{val}\ x \equiv \ell\ \ d_v') : (x{:}\Sigma[\ell], \beta)}$$

$$\frac{\ell \notin \Lambda \qquad \Lambda, \Sigma \vdash (\eta, d_v) \simeq (\eta', d_v') : \beta}{\Lambda, \Sigma \vdash (\eta, \texttt{pointcut}\ x \equiv \texttt{call}(\ell)\ \ d_v) \simeq (\eta', \texttt{pointcut}\ x \equiv \texttt{call}(\ell)\ \ d_v') : (x{:}\texttt{pc}(\Sigma[\ell]), \beta)}$$

$$\frac{\Lambda, \Sigma \vdash (\eta, m_v) \simeq (\eta', m_v') : \sigma \qquad \Lambda, \Sigma \vdash (\eta, d_v) \simeq (\eta', d_v') : \beta}{\Lambda, \Sigma \vdash (\eta, \texttt{structure}\ x \equiv m_v\ \ d_v) \simeq (\eta', \texttt{structure}\ x \equiv m_v'\ \ d_v') : (x{:}\sigma, \beta)}$$

$$\frac{\Lambda, \Sigma \vdash (\eta, d_v) \simeq (\eta', d_v') : \beta}{\Lambda, \Sigma \vdash (\eta, \texttt{struct}\ d_v\ \texttt{end}) \simeq (\eta', \texttt{struct}\ d_v'\ \texttt{end}) : \texttt{sig}\ \beta\ \texttt{end}}$$

$$\frac{\begin{array}{c}\forall m^3, m^4 \quad \emptyset \vdash (\bullet, m^3) \cong (\bullet, m^4) : \sigma' \implies \\ \Lambda, \Sigma \vdash (\eta_1, m_v^1\ m^3) \cong (\eta_2, m_v^2\ m^4) : \sigma\end{array}}{\Lambda, \Sigma \vdash (\eta_1, m_v^1) \simeq (\eta_2, m_v^2) : \sigma' \to \sigma}$$

$$\frac{\forall \ell\ .\ \ell \in (\mathit{domain}(\eta_1) \cup \mathit{domain}(\eta_2)) \wedge \ell \notin \Lambda \implies \Lambda, \Sigma \vdash (\eta_1, \eta_1[\ell]) \simeq (\eta_2, \eta_2[\ell]) : \Sigma[\ell]}{\Lambda, \Sigma \vdash \eta_1 \simeq \eta_2}$$

Fig. 14. Logical Equivalence for Machine Values

recursive calls to `fib`, are hidden in Λ. This is the technical explanation for why the special semantics of `TinyAspect`'s sealing operation are important; without it, all internal calls to the public functions of a module would be available to advice.

This rule also shows the critical importance of keeping advice second-class in `TinyAspect`. In a system with second-class advice, clients can only advise the free labels in a module value, as shown in the rule. If pointcuts and advice were first-class, a function could compute a pointcut dynamically, return it to clients, which could then advise the pointcut. Keeping track of which functions clients could advise would be extremely difficult in this setting.

The rules for logical equivalence of value/environment pairs are defined in Figure 14. In these rules, we consider machine configurations to be equivalent up to alpha-conversion of label names in the environment.

Our rules are similar to typical logical relations rules, but have one important difference. Because `TinyAspect` supports a limited notion of state through the advice mechanism, logical equivalence is defined as a bisimulation [17]. That is, equivalent functions must not only produce equivalent results given equivalent arguments, they must *also* trigger advice on client-accessible labels in the same sequence and with the same arguments. Another way of saying this is that all possibly-infinite traces of pairs of (client-accessible label, argument value) triggered by logically equivalent functions must be themselves equivalent.

This bisimulation cannot be defined inductively on types as is usual for logical relations, because a function of type $\tau \rightarrow \tau'$ may trigger advice on labels whose types are themselves bigger than τ or τ'. Instead, the rules in Figure 14 should be interpreted coinductively for ordinary types–and thus all the rules for ordinary types τ are designed to be monotonic to ensure that the greatest fixpoint of the equivalence relation exists. We can still use an inductive definition of equivalence for module types σ, since module definitions cannot be advised To the best of our knowledge, this coinductive interpretation of logical equivalence rules is novel.

The first rule states that all unit values are equivalent. The second states that we can assume that any non-private label (i.e., one not in Λ) is equivalent to itself. Other labels can be judged equivalent to another value by looking up the label in the environment.

Two functions are equivalent if, when invoked with a fresh label, they execute with that label in a bisimilar way (using the machine expression equivalence judgment from Figure 15). We cannot use the usual logical relations rule for function equivalence, because this rule quantifies over logically equivalent pairs of arguments and is thus non-monotonic and incompatible with our co-inductive definition of equivalence.

Two empty declarations are equivalent to each other. Two `val` declarations are equivalent if they bind the same variable to the same label (since labels are generated fresh for each declaration we can always choose them to be equal when we are proving equivalence). Since the label exposed by the `val` declaration is visible, it must not be in the private set of labels Λ. Pointcut and structure declarations just check the equality of their components. All three declaration forms ensure that subsequent declarations are also equivalent. Two first-order modules are equivalent if the declarations inside them are also equivalent.

For functors, we use the usual logical relations rule: two functors are equivalent if they produce logically-equivalent module results for any logically-equivalent, closed module arguments. This definition is well-formed because equivalence for module types σ is defined inductively rather than coinductively, using the coinduction only for the base case of functions within modules.

Figure 15 shows the rules for logical equivalence of expression/environment pairs. These rules enforce bisimilarity with respect to values returned by a function or functor, or values passed to a non-private label. Our definitions are similar to weak bisimilarity in the π-calculus, with ordinary reductions or lookups of private labels corresponding to τ-transitions in the standard notion of weak bisimilarity.

The first rule states that two expressions are equivalent if they take any number of non-observable steps (written $\mapsto_{\Lambda}^{*}$) to reduce to values that are also equivalent. The non-observable step relation $\rightarrow_{\Lambda}$ is equivalent to ordinary reduction $\rightarrow$, except that the *r-lookup* rule may only be applied to labels in Λ (i.e., those that cannot be advised by clients).

$$\frac{(\eta_1, E_1) \mapsto^*_\Lambda (\eta_1', V_1) \qquad (\eta_2, E_2) \mapsto^*_\Lambda (\eta_2', V_2) \qquad \Lambda, \Sigma \vdash (\eta_1', V_1) \simeq (\eta_2', V_2) : T}{\Lambda, \Sigma \vdash (\eta_1, E_1) \cong (\eta_2, E_2) : T}$$

$$\frac{(\eta_1, E_1) \mapsto^+_\Lambda (\eta_1', E_1') \qquad (\eta_2, E_2) \mapsto^+_\Lambda (\eta_2', E_2') \qquad \Lambda, \Sigma \vdash (\eta_1', E_1') \cong (\eta_2', E_2') : T}{\Lambda, \Sigma \vdash (\eta_1, E_1) \cong (\eta_2, E_2) : T}$$

$$\frac{\begin{array}{cc} \ell, \ell' \notin \Lambda & \Lambda, \Sigma \vdash (\eta_1, v_1) \simeq (\eta_2, v_2) : \tau \\ \Sigma[\ell] = \tau \to \tau' & \Lambda, (\Sigma, \ell' {:} \tau') \vdash (\eta_1, C_1[\ell']) \cong (\eta_2, C_2[\ell']) : T \end{array}}{\Lambda, \Sigma \vdash (\eta_1, C_1[\ell\ v_1]) \cong (\eta_2, C_2[\ell\ v_2]) : T}$$

Fig. 15. Logical Equivalence for Machine Expressions

The second rule allows two expressions to each take one or more non-observable steps (indicated by the $+$ superscript instead of the $*$ superscript for zero or more steps), resulting in observationally-equivalent expressions. Finally, the last rule states that if two equivalent expressions are both at the point where they need to look up a label that is *not* in Λ in order to continue, we must verify that the values to which the labels are applied are also equivalent, and that the contexts are equivalent once a fresh label (representing the result of the application, which is unknowable due to client advice) is substituted into the contexts. Since our definition of equivalence is coinductive, we can use an infinite sequence of the second and third rules to conclude that two non-terminating expressions are logically equivalent.

Now that we have defined logical equivalence, we can state a soundness theorem relating logical and contextual equivalence:

Theorem 3 (Soundness of Logical Equivalence). *If* $E_1 \cong E_2$ *then* $E_1 \equiv E_2$.

For space reasons, we give only a brief sketch of the proof of soundness. More details are available in a companion technical report [1]. The proof proceeds by establishing a bisimulation between two programs that consist of the same context with logically equivalent embedded values. The bisimulation invariant states that the two programs are structurally equivalent except for embedded closed values, which are themselves logically equivalent. The key lemma in the theorem states that the bisimulation is sound; that is, the bisimulation invariant is preserved by reduction.

We then observe that the logically equivalent expressions E_1 and E_2 either both diverge, or both reduce to logically equivalent values. In the former case, any context surrounding the expressions will also diverge, so the expressions are contextually equivalent in this case. In the latter case, the expressions will both reduce to values in the same context, which will then obey the bisimulation invariant described above. The soundness of bisimulation implies that these expressions will either both diverge or will reduce to logically equivalent values. Thus, the expressions are contextually equivalent in this case as well.

5.2 Applying Logical Equivalence

The definition of logical equivalence can be used to ensure that changes to the implementation of one module within an application preserve the application's semantics. For example, consider replacing the recursive implementation of the Fibonacci function in Figure 8 with an implementation based on a loop. In AspectJ, or any module system that does not include the dynamic semantics of our sealing operation, this seemingly innocuous change does not preserve the semantics of the application, because some aspect could be broken by the fact that `fib` no longer calls itself recursively.

Open Modules ensure that this change does not affect the enclosing application, and the logical equivalence rules can be used to prove this. When the module in Figure 8 is sealed, `fib` is bound to a fresh label that forwards external calls to the internal implementation of `fib`. We can show that the two implementations of the module are logically equivalent by showing that no matter what argument value the `fib` function is called with, the function returns the same results and invokes the *external* label in the same way. But the external label is fresh and is unused by either `fib` function, so this reduces to proving ordinary function equivalence, which can easily be done by induction on the argument value. We can then apply the abstraction theorem to show that clients are unaffected by the change.

6 Discussion

In this section, we reconsider the research questions from the introduction in light of the formal model of Open Modules and the logical equivalence definition. Since the formal model can be used to represent the tool support provided by IDEs like the AJDT, we consider how these tools might be enhanced in light of the model.

1. *How can developers specify an interface for a library or component that permits as many uses of advice as possible, while still ensuring critical properties of the component implementation?*

They can do so by declaring explicit pointcuts in the interface of the component that describe "supported" internal events. These pointcuts form a contract between a component provider and a client: The provider promises that if the client obeys the rules of the Open Module system, then the system will have the desired properties, and upgrades to the component will preserve client functionality. The client's side of the contract can be enforced by a compiler, while the component provider's side can be verified using the logical equivalence rules (and their principled extension to a full AOP language)

Once the proper interface has been declared, our logical equivalence rules show how to prove the full functional correctness of a module in a completely modular way, given that an aspect-aware interface has been computed and that an appropriate specification (e.g., in the form of a reference implementation) is available.

Of course, this answer is also limited: we give formal rules for a core language, and full languages are far more complex. However, our system could be used to prove the correctness of compiler optimizations for the constructs that are expressed in the core language. Our system may also be the foundation for a richer set of equivalence rules that can be applied to a full system. Finally, the formal rules in our system may be most helpful by showing engineers how to reason *informally* about the correctness of changes to an aspect-oriented program. Our system yields strong theoretical support to the intuitive notion that changes to a module will not affect clients as long as functions compute the same result and trigger pointcuts in the same way as the original module does.

2. *How can developers specify an interface for a library or component that permits as many uses of advice as possible, while still allowing the component to be changed in meaningful ways without affecting clients?*

The same kind of interface can be defined as in the question above. In the context of software evolution, however, the logical equivalence rules can be used to ensure that a proposed change to a module cannot affect that module's clients, even if the clients themselves are unknown. As long as the client code obeys the Open Module interface, it cannot be broken by an upgraded version of a third-party component.

Note that the client is free to choose to bypass the Open Module rules; a future compiler for Open Modules might issue a warning but compile the code anyway. In this case the client would lose the guarantee that upgrades would preserve the correctness of client code, but gain the ability to reuse or adapt the component in more flexible ways than are permitted by the Open Module system. Depending on the precise circumstances, the reuse benefits might outweigh the potential costs of fixing code that breaks after an upgrade.

Tool Support. As Parnas argued, the primary goal of modularity is to ease software evolution. The model we have developed shows that evolving a module's implementation might also involve changing the pointcuts that act on the module, so that they capture events in the new module's implementation that correspond to the events captured by the original pointcut in the original module specification. Currently the AJDT IDE aids this process by identifying what these pointcuts are, but the pointcuts must still be changed at their definition points, making the implementation task non-local. Our model suggests an improvement to IDEs: providing an editable view of the portion of each pointcut that intersects with a module's source code would allow many changes to be made in a more local way.

Language Design. Our modularity result suggests a number of guidelines for AOP language designers who want to preserve modular reasoning. First, declarative, second-class advice (as in `TinyAspect`) is easier to reason about than first-class advice. Second, in a language with first-class functions, advice should affect function declarations, not the functions themselves, again because this allows a module system to scope the effect of advice. Finally, languages should distinguish advice that affects only the interface of a module from more invasive forms of advice that affect a module's implementation.

7 Related Work

Formal Models. The most closely related formal models are the foundational calculus of Walker et al. [19], and the model of AspectJ by Jagadeesan et al. [10], both of which were discussed in the beginning of Section 3. In other work on formal models of AOP, Lämmel provides a big-step semantics for method-call interception in object-oriented languages [15]. Wand et al. give an untyped, denotational semantics for advice and dynamic join points [20].

Aspects and Modules. Dantas and Walker have extended the calculus of Walker et al. to support a module system [7]. Their type system includes a novel feature for controlling whether advice can read or change the arguments and results of advised functions. In their design, pointcuts are first-class and advice applies to functions, providing more flexibility compared to `TinyAspect`, where pointcuts are second-class and advice applies to declarations. This design choice makes it much more difficult to prove logical equivalence properties, however, because either making pointcuts first-class or tying advice to functions allows join points to escape from a module even if they are not explicitly exported in the module's interface. In their system, functions can only be advised if the function declaration explicitly permits this, and so their system is not oblivious in this respect [8]. In contrast, `TinyAspect` allows advice on all function declarations, and on all functions exported by a module, providing significant "oblivious" extensibility without compromising modular reasoning.

Lieberherr et al. describe Aspectual Collaborations, a construct that allows programmers to write aspects and code in separate modules and then compose them together into a third module [16]. Since they propose a full aspect-oriented language, their system is much richer and more flexible than ours, but its semantics are not formally defined. Their module system does not encapsulate internal calls to exported functions, and thus does not provide as strong an abstraction boundary as Open Modules does.

Other researchers have studied modular reasoning without the use of explicit module systems. For example, Clifton and Leavens propose engineering techniques that reduce dependencies between concerns in aspect-oriented code [5]. Other work has studied analyzing base code and advice separately using interfaces similar to those of Open Modules [14], and analyzing what advice might be affected by a change to code [13].

Our module system is based on that of standard ML [18]. `TinyAspect`'s sealing construct is similar to the freeze operator that is used to close a module to future extensions in module calculi such as Jigsaw [3].

The name Open Modules indicates that modules are open to advice on functions and pointcuts exposed in their interface. Open Classes is a related term indicating that classes are open to the addition of new methods [6].

8 Conclusion

This paper described `TinyAspect`, a minimal core language for reasoning about aspect-oriented programming systems. `TinyAspect` is a source-level language that supports declarative aspects. We have given a small-step operational semantics to the language and proven that its type system is sound. We have described a proposed module system for aspects, formalized the module system as an extension to `TinyAspect`, and proved that the module system enforces abstraction. Abstraction ensures that clients cannot affect or depend on the internal implementation details of a module. As a result, programmers can both separate concerns in their code and reason about those concerns separately.

Acknowledgments

I thank Gregor Kiczales, Mira Mezini, Shriram Krishnamurthi, Ralf Lämmel, Mitch Wand, Karl Lieberherr, David Walker, Tim Halloran, Curtis Clifton, Derek Dreyer, Todd Millstein, Robert Harper, and anonymous reviewers for comments on this material. This work was supported by the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298, NSF grant CCR-0204047, and the Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems."

References

1. J. Aldrich. Open Modules: Modular Reasoning about Advice. Carnegie Mellon Technical Report CMU-ISRI-04-141, available at http://www.cs.cmu.edu/~aldrich/aosd/, Dec. 2004.
2. D. G. Bobrow, L. G. DiMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System Specification. In *SIGPLAN Notices 23*, September 1988.
3. G. Bracha. The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance. Ph.D. Thesis, Dept. of Computer Science, University of Utah, 1992.
4. A. Clement, A. Colyer, and M. Kersten. Aspect-Oriented Programming with AJDT. In *ECOOP Workshop on Analysis of Aspect-Oriented Software*, July 2003.
5. C. Clifton and G. T. Leavens. Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning. In *Foundations of Aspect Languages*, April 2002.
6. C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Object-Oriented Programming Systems, Languages, and Applications*, October 2000.
7. D. S. Dantas and D. Walker. Aspects, Information Hiding and Modularity. Princeton University Technical Report TR-696-04, 2004.
8. R. E. Filman and D. P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *Advanced Separation of Concerns*, October 2000.
9. S. Gudmundson and G. Kiczales. Addressing Practical Software Development Issues in AspectJ with a Pointcut Interface. In *Advanced Separation of Concerns*, July 2001.

10. R. Jagadeesan, A. Jeffrey, and J. Riely. An Untyped Calculus of Aspect-Oriented Programs. In *European Conference on Object-Oriented Programming*, July 2003.
11. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *European Conference on Object-Oriented Programming*, June 2001.
12. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming*, June 1997.
13. C. Koppen and M. Stoerzer. PCDiff: Attacking the Fragile Pointcut Problem. In *European Interactive Workshop on Aspects in Software*, September 2004.
14. S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying Aspect Advice Modularly. In *Foundations of Software Engineering*, Nov. 2004.
15. R. Lämmel. A Semantical Approach to Method-Call Interception. In *Aspect-Oriented Software Development*, Apr. 2002.
16. K. Lieberherr, D. H. Lorenz, and J. Ovlinger. Aspectual Collaborations: Combining Modules and Aspects. *The Computer Journal*, 46(5):542–565, September 2003.
17. R. Milner. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, 1999.
18. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997.
19. D. Walker, S. Zdancewic, and J. Ligatti. A Theory of Aspects. In *International Conference on Functional Programming*, 2003.
20. M. Wand, G. Kiczales, and C. Dutchyn. A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming. *Transactions on Programming Languages and Systems*, 26(5):890–910, September 2004.

Evaluating Support for Features in Advanced Modularization Technologies

Roberto E. Lopez-Herrejon, Don Batory, and William Cook

Department of Computer Sciences, University of Texas at Austin,
Austin, Texas, 78712, U.S.A
{rlopez, batory, wcook}@cs.utexas.edu

Abstract. A *software product-line* is a family of related programs. Each program is defined by a unique combination of features, where a *feature* is an increment in program functionality. Modularizing features is difficult, as feature-specific code often cuts across class boundaries. New modularization technologies have been proposed in recent years, but their support for feature modules has not been thoroughly examined. In this paper, we propose a variant of the expression problem as a canonical problem in product-line design. The problem reveals a set of technology-independent properties that feature modules should exhibit. We use these properties to evaluate five technologies: AspectJ, Hyper/J, Jiazzi, Scala, and AHEAD. The results suggest an abstract model of feature composition that is technology-independent and that relates compositional reasoning with algebraic reasoning[1].

1 Introduction

A *feature* is an increment in program functionality [53]. Researchers in software product-lines use features as a defacto standard in distinguishing the individual programs in a product-line, since each program is defined by a unique combination of features [24]. Features are the semantic building blocks of program construction; a product-line model is a set of features and constraints among features that define legal and illegal combinations. Product-line architects reason about programs in terms of features.

Despite their crucial importance, features are rarely modularized. The reason is that feature-specific code often cuts across class and package boundaries, thus requiring the use of preprocessors to wrap feature-specific code fragments in `#if-#endif` statements. While the use of preprocessors works in practice, it is hardly an adequate substitute for proper programming language support. Among the important properties sacrificed are: static typing of feature modules, separate compilation of feature modules, and specifications of feature modules that are independent of the compositions in which they are used (a property critical for reusability). This sacrifice is unacceptable.

In recent years, new technologies have been proposed that have the potential to provide better support for feature modularity. These technologies have very different

[1] This research is sponsored in part by NSF's Science of Design Project #CCF-0438786.

notions of modularity and composition, and as a consequence are difficult to compare and unify. Thus it is increasingly important to advance standard problems and metrics for technology evaluation. A few attempts have been made to compare technologies and evaluate their use to refactor and re-implement systems that are not part of a product family [13][19][30][37]. But for a few studies [16][52][35], the use of new technologies to modularize features in a product line is largely unexplored.

In this paper we present a standard problem that exposes common and fundamental issues that are encountered in feature modularity in product-lines. The problem reveals technology-independent properties that feature modules should exhibit. We use these properties to evaluate solutions written in five novel modularization technologies: AspectJ [1][25], Hyper/J [41][48], Jiazzi [31][32][52], Scala [45][38][39][40], and AHEAD [2][6]. The results suggest a technology-independent model of software composition where the definition and composition of features is governed by algebraic laws. The model provides a framework or set of criteria that a rigorous mathematical presentation should satisfy. Further, it helps reorient the focus on clean and mathematically justifiable abstractions when developing new tool-specific concepts.

2 A Standard Problem: The Expressions Product-Line

The *Expressions Product-Line (EPL)* is based on the extensibility problem also known as the "expression problem" [15][50]. It is a fundamental problem of software design that consists of extending a data abstraction to support a mix of new operations and data representations. It has been widely studied within the context of programming language design, where the focus is achieving data type and operation extensibility in a type-safe manner. Rather than concentrating on that issue, we consider the *design and synthesis* aspects of the problem to produce a family of program variations. More concretely, what features are present in the problem? How can they be modularized? And how can they be composed to build all the programs of the product-line?

2.1 Problem Description

Our product-line is based on Torgersen's expression problem [49]. Our goal is to define data types to represent expressions of the following language:

```
Exp  :: = Lit | Add | Neg
Lit  :: = <non-negative integers>
Add  :: = Exp "+" Exp
Neg  :: = "-" Exp
```

Two operations can be performed on expressions of this grammar:

1) `Print` displays the string value of an expression. The expression `2+3` is represented as a three-node tree with an `Add` node as the root and two `Lit` nodes as leaves. The operation `Print`, applied to this tree, displays the string "`2+3`".
2) `Eval` evaluates expressions and returns their numeric value. Applying the operation Eval to the tree of expression `2+3` yields 5 as result.

We add a class `Test` that creates instances of the data type classes and invokes their operations. We include this class to demonstrate additional properties that are important for feature modules. Figure 1 shows the complete Java code for a program of the product-line that implements all the data types and operations of EPL. Shortly we will see what the annotations at the beginning of each line mean.

```
lp  interface Exp {
lp    void print();
le    int eval();
lp  }

ap  class Add implements Exp {
ap    Exp left, right;
ap    Add (Exp l, Exp r) {
ap     left = l; right = r; }
ap    void print() {
ap      left.print();
ap      System.out.print("+");
ap      right.print();
ap    }
ae    int eval() {
ae      return left.eval()
ae           + right.eval();
ae    }
ap  }

np  class Neg implements Exp {
np    Exp expr;
np    Neg (Exp e) { expr = e; }
np    void print() {
np      System.out.print("-(");
np      expr.print();
np      System.out.print(")");
np    }
ne    int eval() {
ne      return expr.eval() * -1;
ne    }
np  }
```

```
lp  class Lit implements Exp {
lp    int value;
lp    Lit (int v) { value = v; }
lp    void print() {
lp      System.out.print(value);
lp    }
le    int eval() { return value; }
lp  }

lp  class Test {
lp    Lit ltree;
ap    Add atree;
np    Neg ntree;
lp    Test() {
lp      ltree = new Lit(3);
ap      atree = new Add(ltree, ltree);
np      ntree = new Neg(ltree);
lp    }
lp    void run() {
lp      ltree.print();
ap      atree.print();
np      ntree.print();
le      System.out.println(ltree.eval());
ae      System.out.println(atree.eval());
ne      System.out.println(ntree.eval());
lp    }
lp  }
```

Fig. 1. Complete code of the Expressions Product Line

From a product-line perspective, we can identify two different feature sets [17]. The first is that of the operations {`Print, Eval`}, and the second is that of the data types {`Lit`, `Add`, `Neg`}. Using these sets, it is possible to synthesize all members of the product-line described in Figure 2 by selecting one or more operations, and one or more data types. For instance, row 4 is the program that contains `Lit` and `Add` with operations `Print` and `Eval`. As with any product-line design, in EPL there are constraints

	Operations		Data types		
Program	Print	Eval	Lit	Add	Neg
1	✓		✓		
2	✓	✓	✓		
3	✓		✓	✓	
4	✓	✓	✓	✓	
5	✓		✓		✓
6	✓	✓	✓		✓
7	✓		✓	✓	✓
8	✓	✓	✓	✓	✓

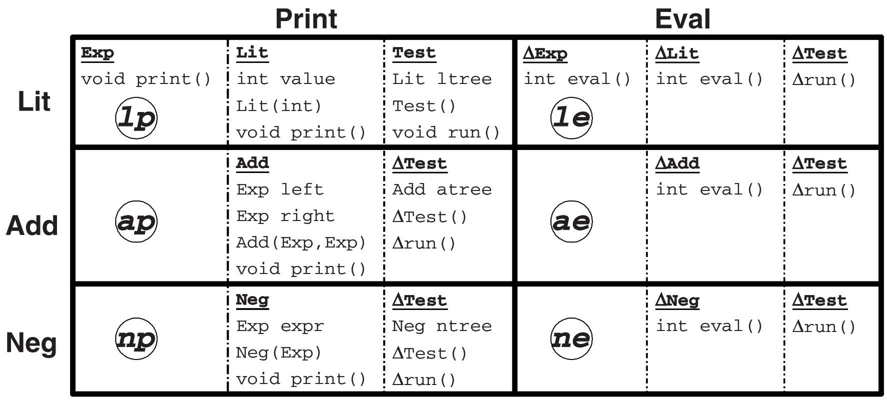
Fig. 2. Members of the EPL

on how features are combined to form programs. For example, all members require `Lit` data type, as literals are the only way to express numbers.

A common way to implement features in software product-lines is to use preprocessor declarations to surround the lines of code that are specific to a feature. If we did this for the program in Figure 1, the result would be unreadable. Instead, we use an annotation at the start of each line to indicate the feature to which the line belongs. This makes it easy to build a preprocessor that receives as input the names of the desired features and strips off from the code of Figure 1 all the lines that belong to unneeded features. As can be imagined, this approach is very brittle for problems of larger scale and complexity. Never the less, the approach can be used as a reference to define what is expected from feature modules in terms of functionality (classes, interfaces, fields, methods, constructors), behaviour (sequence of statements executed), and composition.

2.2 Feature Modularization

A natural representation of the expression problem, and thus for EPL, is a two-dimensional matrix [15][50][22]. The vertical dimension specifies data types and the horizontal dimension specifies operations. Each matrix entry is a *feature module* that implements the operation, described by the column, on the data type, specified by the row. As a naming convention throughout the paper, we identify matrix entries by using the first letters of the row and the column, e.g., the entry at the intersection of row `Add` and column `Print` is named `ap` and implements operation `Print` on data type `Add`. This matrix is shown in Figure 3 where module names are encircled.



	Print			Eval		
Lit	**Exp** void print() 1p	**Lit** int value Lit(int) void print()	**Test** Lit ltree Test() void run()	**ΔExp** int eval() 1e	**ΔLit** int eval()	**ΔTest** Δrun()

Add	ap	**Add**
		Exp left
		Exp right
		Add(Exp,Exp)
		void print()
			ΔTest
			Add atree
			ΔTest()
			Δrun()
				ae	**ΔAdd**
					int eval()
						ΔTest
						Δrun()

Neg	np	**Neg**
		Exp expr
		Neg(Exp)
		void print()
			ΔTest
			Neg ntree
			ΔTest()
			Δrun()
				ne	**ΔNeg**
					int eval()
						ΔTest
						Δrun()

Fig. 3. Matrix representation and Requirements

To compose any program from Figure 2, the modules involved are those at the intersection of the selected columns and the selected rows. For example, program number 1, that provides `Print` operation on `Lit`, only requires module `lp`. Another example is program 6, that implements operations `Print` and `Eval` on `Lit` and `Neg` data types, requires modules `lp`, `le`, `np`, and `ne`.

The source code of a feature module are the lines that are annotated with the name of the module. For instance, the contents of feature `ap` include:

a) Class `Add` with `Exp` fields `left` and `right`, a constructor with two `Exp` arguments, and method `void print()`, and
b) An increment to class `Test`, because it is adding something to the class as opposed to contributing a brand new class as is the case of class `Add`. This increment is symbolized by `ΔTest` in Figure 3. It adds: field `atree`, a statement to the body of the constructor expressed with `ΔTest()`, and a statement to the body of method `run` expressed as `Δrun()`.

For clarity we decided to put the `Exp` interface inside module `lp` instead of creating a separate row for it. This decision makes sense since the other data types are built using `Lit` objects. Also, we put the constructors and fields of the data types in column `Print` instead of refactoring them into a new column and have columns `Print` and `Eval` implement only their corresponding methods. Later we will see an interesting consequence of these two design decisions. Additionally, from the design requirements we can infer dependencies and interactions among the feature modules. For instance, if we want to build a program with module `ap`, we also need to include module `lp` because `ap` increments the `Test` class which is introduced in `lp`. Later, we briefly discuss this issue as compositional constraints, which are not the focus of this paper. Constraints are discussed in [2][3][9].

3 Basic Properties for Feature Modularity

To give structure to our evaluation, we identify a set of basic properties about features that can readily be inferred from, illustrated by, and assessed in EPL and its solutions in the five technologies evaluated. Conceivably, there are other desirable properties that feature modules should exhibit such as readability, ease of use, etc. However, for sake of simplicity and breadth of scope, they are not part of this evaluation as their objective assessment would require a larger case study that would prevent us from comparing all five technologies together.

The properties are grouped into two categories, covering the basic definition of features and their composition to create programs. The first properties in each category follow from the structure of EPL, while the others come from the studied solutions to EPL and are desirable from the software engineering perspective.

3.1 Feature Definition Properties

The first category of properties relate to the definition of the basic building blocks of EPL, the representation of each piece, and their organization into features.

Program Deltas. The code in Figure 1 can be decomposed into a collection of *program deltas* or program fragments. The kinds of program deltas required to solve EPL are summarized in Figure 3, and include:

- *New Classes*, for example `Lit` in module `lp`.
- *New Interfaces*, for example `Exp` in module `lp`.
- *New fields* that are added to existing classes, like field `atree` in module `ap` is added to class `Test`.
- *New methods* that are added to existing interfaces, like `eval()` in module `le` is added to interface `Exp`.
- *Method extensions* that add statements to methods. For example, extension to method `run()`, expressed by Δ`run()`, in all modules except `lp`.
- *Constructor extensions* that add statements to constructors. For instance, extensions to constructor `Test()`, expressed by `ΔTest()`, in modules `ap` and `np`.

There are other program deltas, such as new constructors, new static initializers, new exception handlers, etc. that are not needed for implementing EPL and thus are not considered in this paper. Nonetheless, we believe that EPL contains a sufficient set of program deltas for an effective evaluation.

Cohesion. It must be possible to collect a set of program deltas and assign them a name so that they can be identified and manipulated as a cohesive module.

Separate Compilation. Separate compilation of features is useful for two practical reasons: a) it allows debugging of feature implementation (catching syntax errors) in isolation, and b) it permits the distribution of bytecode instead of source code.

3.2 Feature Composition Properties

Once a set of feature modules has been defined, it must be possible to compose them to build all the specific programs in the Expression Product Line.

Flexible Composition. The implementation of a feature module should be syntactically independent of the composition in which it is used. In other words, a fixed composition should not be hardwired into a feature module. Flexible composition improves reusability of modules for constructing a family of programs.

Flexible Order. The order in which features are composed can affect the resulting program. For instance, in EPL, the order of test statements in method `run()` affects the output of the program. The program in Figure 1 is the result of one possible ordering of features, namely (`lp`, `ap`, `np`, `le`, `ae`, `ne`). Another plausible order in EPL is to have expressions printed and evaluated consecutively, as in order (`lp`, `le`, `ap`, `ae`, `np`, `ne`). Hence, feature modules should be composable in different orders.

Closure Under Composition. Feature modules are closed under composition if one or more features can be composed to make a new composite feature. Composite features must be usable in all contexts where basic features are allowed. In EPL, it would be natural to compose the `Lit` and `Neg` representations to form a `LitNeg` feature which represents positive and negative numbers.

Static Typing. Feature modules and their composition are subject to static typing which helps to ensure that both are well-defined, for example, preventing method-not-found errors. We base the evaluation of this property on the availability of a formal typing theory or mechanism behind each technology.

Using these properties we evaluate AspectJ, Hyper/J, Jiazzi, Scala, and AHEAD in the following sections.[2] We use a concrete example to illustrate these alternatives, i.e. the program that supports `Print` and `Eval` operations in `Lit` and `Add` data types (program number 4 in Figure 2). Thus, the program has four modules: `lp`, `ap`, `le`, and `ae` that we compose in this order (the same as in Figure 1). Throughout the paper, we call this program `LitAdd`.

4 AspectJ

An aspect, as implemented in *AspectJ*[3] [1][25], modularizes a *cross-cut* as it contains code that can extend several classes and interfaces.

4.1 Feature Modules and Their Composition

The implementation of module `lp` is straightforward as it consists of Java interface `Exp` and classes `Lit` and `Test`. In AspectJ literature, programs written using only pure Java code are called *base code*. In Figure 4a, the names of files that are base code are shown in italics, while those of *aspect code* are shown in all capital letters.

	Print	Eval
Lit	*Exp, Lit, Test*	LE
Add	*Add*, AP	AE
Neg	*Neg*, NP	NE

(a)

```
public aspect LE {

  // ΔExp interface
  public abstract int Exp.eval();

  // ΔLit class
  public int Lit.eval() { return value; }

  // ΔTest, advice that implements Δrun()
  pointcut LPRun(Test t):
     execution (public void Test.run())
     && target(t);

  void around(Test t) : LPRun(t) {
    proceed(t);
    System.out.println("= "  + t.ltree.eval());
  }
}
```

(b)

Fig. 4. AspectJ Solution

[2] For a more detailed description of the implementation see [29].

[3] We used AspectJ version 1.1 for our evaluation.

Alternatively, we could have declared the new classes and interfaces as nested elements of an aspect. However, they would be subject to the instantiation of their containing aspect, and their references would be qualified with the aspect name where they are declared. For these reasons, we decided to implement classes and interfaces in separate files.

From Figure 3, module `le`:

1) adds method `eval()` to interface `Exp`,
2) adds the implementation of `eval()` to class `Lit`, and
3) appends a statement to method `run()` of class `Test` that calls `eval()` on field `ltree`.

The entire code of module `le` is implemented with the aspect shown in Figure 4b. The first two requirements use AspectJ's *inter-type declaration*, which is part of its *static crosscutting* model [1][25][4]. Method extensions, like that of the third requirement, cannot be implemented as inter-type declarations because members with the same signature can be introduced only once. Hence, to implement the last requirement it is necessary to utilize AspectJ's *dynamic crosscutting* model which permits adding code (*advice*) at particular points in the execution of a program (*join points*) that are specified through a predicate (*pointcut*).

Since it is required to execute an additional statement when method `run()` is executed, we must capture the join point of the `execution` of that method. Also, since the statement to add is a method call on field `ltree` of class `Test`, we must get a hold of the object that is the `target` of the execution of method `run()` to access its `ltree` field. These two conditions are expressed in pointcut `LPRun` of Figure 4b, where `t` is the reference to the target object. Lastly, to add the extension statement we use an `around` advice. This type of advice executes instead of the join points of the pointcut, but it allows its execution by calling AspectJ's special method `proceed`. We add the new statement to `run()` after the call to method `proceed(t)`.[5]

The implementation of feature module `ap` (not shown in Figure 4) uses two files. The first is a Java class to implement data type `Add`. The second is an aspect to implement the extensions to class `Test`. The first extension adds a new field to class `Test`. This is done also using inter-type declaration in the following way:

```
public Add Test.atree;
```

The other two extensions of module `ap`, `ΔTest()` and `Δrun()`, are implemented in a similar way to those of module `le`. The other modules `ae`, `np`, and `ne` have an analogous implementation.

To compose program `LitAdd`, the AspectJ compiler (*weaver*) `ajc`, requires the file names of the base code and the aspects of the feature modules. The composition is specified as follows, where the order of the terms is inconsequential:

---

[4] We could also implement the first requirement as follows:

```
public int Exp.eval() { return 0; }
```

This alternative defines a default value for the method which can be subsequently overridden by each class that implements Exp.

[5] Method `proceed`, has the same arguments as the advice where it is used.

```
ajc Exp.java Lit.java Test.java LE.java Add.java AP.java AE.java
   -outjar LitAdd.jar                                            (1)
```

The static crosscutting model of AspectJ has a simple realization that does not depend on order, namely, members can only be introduced once. However, in the case of dynamic crosscutting, i.e. pointcuts and advice, several pieces of advice can apply to the same join point. In such cases, the order in which advice code is executed is in general undefined[6]. This means that a programmer cannot know a priori, by simply looking at the pointcut and advice code, in what order advice is applied. In program `LitAdd`, this issue is manifested in the order of execution of method `run()` and its extensions. The order that we want is that of Figure 1, namely, first the statement from `lp` followed by those of `ap`, `le` and `ae`. However, the order obtained by executing the program is statements from `lp`, `ae`, `ap`, and `le`[7].

AspectJ provides a mechanism to give precedence to advice, thus imposing an order, at the aspect level. In other words, it can give precedence to all the advice of an aspect over those of other aspects. To obtain the order that we want for method `run()`, we must define the following aspect:

```
public aspect Ordering {
       declare precedence : AE, LE, AP;
}
```

and add it to the list of files in the specification (1). For further details on how precedence clauses are built, consult [1][26].

4.2 Evaluation

Feature Definition. AspectJ can describe all program deltas required for EPL. However, in cases like module `ap` which is implemented with class `Add` and aspect `AP`, there is no way to express that both together form feature `ap`. In other words, AspectJ does not have a cohesion mechanism to group all program deltas together and manipulate them as a single module. Nonetheless, this issue can be addressed with relatively simple tool support. Aspects cannot be compiled separately, as they need have base code in which to be woven.

Feature Composition. AspectJ provides flexible composition and order. It can be used to build all members of EPL in the order described in an auxiliary aspect that contains a `declare precedence` clause. This type of clause can also be used inside aspects that implement feature modules, like `LE`, but doing that could reduce order flexibility as the order could be different for different programs where `LE` is used. Feature modules implemented in AspectJ are not closed under composition for two reasons: the absence of a cohesion mechanism and the lack of a general model of aspect composition. The latter is subject of intensive research [18]. Static typing support for AspectJ is also an area of active research [23][51].

[6] There are special rules that apply for certain types of advice when advices are defined in either the same aspect or in others [1]. These rules help determine the order in few cases but not in general.

[7] In AspectJ version 1.1.

5 Hyper/J

Hyper/J [48] is the Java implementation of an approach to *Multi-Dimensional Separation of Concerns* (*MDSoC*) called Hyperspaces [41][47]. A *hyperspace* is a set of units. A unit can be either primitive, such as a field, method, and constructor; or compound such as a classe, interface, and package.

A *hyperslice* is a modularization mechanism that groups all the units that implement a *concern* (a feature in this paper) which consists of a set of classes and interfaces. Hyperslices must be *declaratively complete*. They must have a declaration, that can be incomplete (stub) or abstract, for any unit they reference. Hyperslices are integrated in *hypermodules* to build larger hyperslices or even complete systems.

5.1 Feature Modules and Their Composition

The Hyper/J weaver performs composition at the bytecode level which makes a natural decision to implement each hyperslice (feature module) as a package that can be compiled independently. Hyperslices that contain only new classes and interfaces, like module `lp`, have a straightforward implementation as Java packages. The interesting case is hyperslices that extend units in other hyperslices. For example, Figure 5a shows the package that implements feature `le`. It adds method `eval()` to `Exp` (new method in an interface), the implementation in `Lit` (new method in a class), and a call in method `run()` of class `Test` (method extension).

```
interface Exp {
   int eval();
}
```

```
class Lit implements Exp {
  public int value;  // stub lp
  public Lit (int v) { } // req constructor
  public int eval() { return value; }
}
```

```
class Test {
  Lit ltree; // stub lp
  public void run() {
   System.out.println(ltree.eval());
  }
}
```

(a) Package LE of feature le

```
Hyperspace (hs)
hyperspace  LitAdd
 composable class LP.*;
 composable class LE.*;
 composable class AP.*;
 composable class AE.*;
```

```
Concern Mapping (cm)
package LP : Feature.LP
package LE : Feature.LE
package AP : Feature.AP
package AE : Feature.AE
```

```
Hypermodule (hm)
hypermodule LitAdd
 hyperslices:
   Feature.LP,
   Feature.AP,
   Feature.LE,
   Feature.AE;
 relationships:
   mergeByName;
end hypermodule;
```

(b) Composition Specification

Fig. 5. Hyper/J Implementation

However, extra code is required to make a hyperslice declaratively complete so that it can be compiled. For instance, variable `value` that is introduced in feature `lp` is replicated in class `Lit` so that it can be returned by method `eval()`. Something similar occurs with variable `ltree` in `Test`. Additionally, the Hyper/J weaver requires stubs for non-default constructors. When the package is compiled, the references of these variables are bound to the definitions in the package; however, when composed with other hyperslices that also declare these variables, all the references are bound to a single declaration determined by the composition specification. The extension of methods and constructors is realized by appending the code of their bodies one after the other. The rest of the feature modules are implemented similarly.

The `LitAdd` composition is defined by the three files of Figure 5b: hyperspace `Lit-Add.hs`, concern mapping `LitAdd.cm`, and hypermodule `LitAdd.hm`. The hyperspace file lists all the units that participate in the composition. The concern mapping divides the hyperspace into features (hyperslices) and gives them names. Finally, the hypermodule specifies what hyperslices are composed and what mechanisms (operators) to use. Our example merges units that have the same name.

5.2 Evaluation

Feature Definition. Hyper/J's hyperslices can modularize all deltas, treat them as a cohesive unit, and compile them separately. Though, separate compilation requires manual completion of the hyperslices.

Feature Composition. Hyper/J provides flexible composition. The order is specified in the hypermodule and can be done using several composition operators [48], thus composition order is flexible. Hyperslices are by definition closed under composition. To the best of our knowledge there is no theory to support static typing of hyperslices.

6 Jiazzi

Jiazzi [31][32][52] is a component system that implements units [21][22] in Java. A *unit* is a container of classes and interfaces. There are two types of units: *atoms*, built from Java programs, and *compounds* built from atoms and other compounds. Units are the modularization mechanism of Jiazzi. Therefore they are the focus of our evaluation.

6.1 Feature Modules and Their Composition

Jiazzi programs use pure Java constructs. Jiazzi groups classes and interfaces in *packages* that are syntactically identical to Java packages. Implementation of modules like `lp` are thus standard Java packages with normal classes and interfaces. Consider the following code contained in package `le` that implements the feature of the same name[8]:

[8] Definition of non-default constructors is required but not shown.

```
public interface Exp extends lp.Exp {
     int eval();
}
public class Lit extends lp.Lit
 implements fixed.Exp {
    public int eval() { return value; }
}
public class Test extends lp.Test {
  public void run() {
    super.run();
    System.out.println(ltree.eval());
   }
}
```

Two important things to note are: a) `Exp`, `Lit` and `Test` *extend* their counterparts of feature `lp`, and b) class `Lit` implements `fixed.Exp` which refers to the version of `Exp` that contains all the extensions in a composition.

Package `le` shows how methods can be added to existing classes and interfaces, and how existing methods can be extended. Jiazzi also supports adding new classes, interfaces, constructor extensions, and fields in a similar way to that of normal Java inheritance. The rest of the feature modules are implemented along the lines of module `le`.

Composition in Jiazzi is elaborate. For simplicity, we illustrate unit composition with units `lp` and `le` instead of `LitAdd`. From this readers can infer what the composition of `LitAdd` entails.

We start with the definition of a *signature* which describes the structure of a package, i.e., the interface it exports. The following code is the signature of package `le`[9]:

```
signature leS = l : lpS + {
  package fixed;
  public interface Exp { int eval(); }
  public class Lit { public int eval(); }
  public class Test { public void run(); }
}
```

Two relevant points are: a) the expression `l:lpS +` indicates that `leS` is an extension of signature `lpS`, meaning that `Exp`, `Lit`, and `Test` of `le` extend their counterparts in `lp`, and b) `fixed` is a *package parameter* that is used, as we have seen, in the implementation of `le`. How this parameter is bound is explained shortly.

A unit definition consists of import and export packages followed (if necessary) by a series of statements that establish relations among the packages which, in the case of compound units, determines the order in which units are composed. Each of the features in our problem is implemented by an atom, and a program in the EPL is expressed by a compound unit. The following code defines unit `le`:

```
atom le {
  import lp : lpS;
  export le extends lp : leS;
  import fixed extends le;
}
```

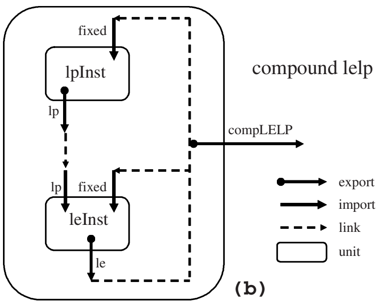
[9] For convention in this section, we form signature names with the names of the packages they described followed by a S.

It asserts that atom `le` imports package `lp` with signature `lpS` and that it exports package `le` of signature `leS` which is an extension of `lp`. It also states that it imports package `fixed`, an extension of `le` which is bound, at composition time, to the package parameter of the same name in the signature.

Jiazzi supports composition through the *Open Class Pattern* [31][32]. The key element of this pattern is the creation of a package, called fixed in our example, that contains all the extensions made by the units. This package is imported by the atom units, creating a feedback loop that permits them to refer to the most extended version of the classes and interfaces involved in a composition.

```
compound lelp {
  export compLELP : leS;                        (1)
  bind package compLELP to compLELP@fixed;  (2)
  }{
    link unit lpInst : lp, leInst : le;      (3)
    link package
        leInst@le to *@fixed,                (4)
        lpInst@lp to leInst@lp,              (5)
        leInst@le to compLELP;               (6)
}
```

(a)



(b)

Fig. 6. Jiazzi Composition of `le` and `lp`

Figure 6a shows the code that composes these two units. Figure 6b illustrates this composition. Consider the second part of the specification first. It states that the composition contains two units (line 3): `lpInst` an instance of unit `lp`, and `leInst` an instance of unit `le`. The packages of these two units are linked as follows: a) line 4 states that the exported package `le` of `leInst` is bound to all the `fixed` packages in the compound, b) line 5 sets the link between the export package `lp` of `lpInst` to the import package `lp` of `leInst`.

To be useful, compound packages must export something, in our case it exports a package that we named `compLELP` with signature `leS` (line 1) which is linked to package `le` of unit `leInst` in line 6. Since `compLELP` has signature `leS` that contains package parameter `fixed` we must bind it, in this case to itself, as done in line 2.

Signatures allow separate unit compilation. Jiazzi provides a stub generator that uses the unit's signature to create the packages and the code skeletons of the classes and interfaces required to compile the unit. It also provides a linker that checks that the compiled unit conforms to the unit's signature and stores the unit's binaries and signature into a Java archive (`jar`) file that can be used to compose with other units. For further details on the stub generator and linker refer to [33].

6.2 Evaluation

Feature Definition. Jiazzi units can modularize all program deltas of EPL in a cohesive way. Furthermore, signatures allow separate compilation.

Feature Composition. Jiazzi separates clearly the implementation of features from their composition thus provides a flexible composition. The order of unit composition is determined by the linking statements in compound units and therefore it is flexible. By definition, units are closed under composition. Jiazzi is backed up with a formal theory for type checking units and their compositions [21][22]. This theory permits the linker to statically check and report errors in program composition.

Jiazzi's type checking and separate compilation come with a price. Defining signatures and wiring the relationships between units is a non-trivial task, especially when dealing with multiple units with complex relations among them [52].

7 Scala

Scala is a strongly-typed language that fuses concepts from object-oriented programming and functional programming [45][38]. Though Scala borrows from Java, it is not an extension of it. We included Scala[10] in our evaluation because it supports two nontraditional modularization mechanisms: traits [44] and mixins [10].

7.1 Feature Modules and Their Composition

A trait in Scala can be regarded as an abstract class without state and parameterized constructors. It can implement methods and contain inner classes and traits. We implemented each feature module by a trait. Consider the implementation of feature `lp` shown in Figure 7a. The trait contains:

- Abstract type `exp` with upper bound `Exp`. This means that `exp` is at least a subtype of `Exp` and thus it leaves `exp` open for further extensions by other features.
- Trait `Exp` declares method `print()`. A trait is used in this context because it is roughly equivalent to a Java interface, as it declares a type with methods whose implementations are not yet defined.
- Class `Lit` extends `Exp`.[11] It has a *primary constructor* (or main constructor) that receives an integer which is assigned to field `value`. It also provides an implementation for method `print()` that displays this field.
- Class `Test` contains abstract field `ltree` of abstract type `exp`. Because of this, class `Test` is also `abstract`. `Test` also contains method `run()` that calls method `print()` on `ltree`.

Trait `ap` is implemented as an extension of trait `lp` , shown in Figure 7b, that contains:

- Class `Add` that extends trait `Exp` of module `lp`. It has a two parameter constructor to initialize the expression fields and the implementation of method `print()`.

[10] We used version 1.3.0.10 for our evaluation.

[11] Scala traits are conceptually not different from classes so that is why we use an extends clause instead of implements.

```
package epl;
trait lp {
  type exp <: Exp;
  trait Exp {
    def print(): unit;
  }
  class Lit(v: int) extends Exp {
    val value = v;
    def print(): unit = System.out.print(value);
  }
  abstract class Test {
    val ltree: exp;
    def run(): unit = { ltree.print(); }
  }
}
```
(a)

```
 package epl;
trait le extends lp {
  type exp <: Exp;
  trait Exp extends super.Exp {
    def eval(): int
  }
  class Lit(v: int) extends super.Lit(v) with Exp {
    def eval(): int = value;
  }
  abstract class Test extends super.Test {
    override def run(): unit = {
      super.run();
      System.out.println(ltree.eval());
    }
  }
}
```
(c)

```
package epl;
trait ap extends lp {
  class Add(l: exp, r: exp) extends super.Exp {
    val left = l; val right = r;
    def print(): unit = {
      left.print(); System.out.print("+");
     right.print();
    }
  }
  abstract class Test extends super.Test {
    val atree: exp;
    override def run(): unit = {
      super.run(); atree.print();
    }
  }
}
```
(b)

```
package epl;
trait ae extends ap with le {
  class Add(l: exp, r: exp) extends super.Add(l, r)
                                    with Exp   {
    def eval(): int = left.eval() + right.eval()
  }

  abstract class Test extends super.Test {
    override def run(): unit = {
      super.run();
      System.out.println(atree.eval());
    }
  }
}
```
(d)

```
package epl;
abstract class Test1 extends lp with ap {
  abstract class Test extends super.Test with super[ap].Test;
}
abstract class Test2 extends Test1 with le {
  abstract class Test extends super.Test with super[le].Test;
}
abstract class Test3 extends Test2 with ae {
  abstract class Test extends super.Test with super[ae].Test;
}
object LitAddObj extends Test3 {
  type exp = Exp;
  class Test  extends super.Test {
    val ltree = new Lit(3);
    val atree = new Add(ltree, new Lit(7));
  }
  def main(args: Array[String]) : unit = {
    var test = new Test();
    test.run();
  }
}
```
(e)

Fig. 7. Scala Solution

- Extension to class `Test`, that adds field `atree` and extends method `run()` with the call to `print()` on this field [12]. This class is also `abstract` because `atree`'s type is abstract.

[12] To prevent inadvertent overriding, Scala requires overriding methods to include an `override` modifier as part of their definitions. Notice also that the overridden method can still be called using super as in Java.

Trait `le` is also implemented as an extension to trait `lp` and is shown in Figure 7c. This trait has:

- Trait `Exp` extends `Exp` of feature `lp` by adding method `eval()`.
- Abstract type `exp` that extends `exp` of feature `lp`, meaning that `exp` is now at least a subtype of `Exp` that has `print()` and `eval()` methods.
- An extension of class `Lit`. This class uses *mixin composition* (expressed as `with Exp` in the figure) to indicate that `Lit` is also a subtype of `Exp` and thus it must implement both of its methods. Since it inherits `print()` from trait `lp` it only needs to implement `eval()`.
- An extension of class `Test` that modifies `run()` to invoke `eval()` on `ltree`.

Feature `ae` is implemented as an extension of feature `ap` and a mixin composition with feature `le` because it provides an implementation of method `eval()` for class `Add`. The code is shown in Figure 7d. Additionally this trait extends method `run()` of class `Test`. The other two feature modules of EPL, `np` and `ne`, are implemented similarly.

To define program `LitAdd` is necessary to: a) specify the order in which method extensions are composed, and b) to create an `object`, a singleton object of a new class, to run the program. Figure 7e illustrates this. For the first part, we use *deep mixin composition* [54] (mixin composition at trait level and nested class level), to establish a linear order of `Test` classes as they contain extensions of method `run()`. For the second part, we define `LitAddObj` that extends `Test3` (the most refined abstract `Test` class), binds abstract type `exp` to concrete type `Exp` as defined by `Test3`, and makes concrete class `Test` by creating instances for the test objects `ltree` and `atree`. The main method creates an instance of `Test` and calls method `run()` on it.

7.2 Evaluation

Feature Definition. Scala can implement all program deltas of EPL. Regarding cohesion, traits provide a mechanism to collect program deltas under a single name. Separate compilation in Scala requires traits and classes to be placed in named packages, as it is illustrated by package `epl` in Figure 7.

Feature Composition. Scala provides flexible composition and flexible order mechanism for implementing EPL. Scala uses inheritance and mixin composition to compose program deltas that add new classes, traits, fields, methods and simple constructor extensions. However, specifying the order of method extensions is a verbose and non-trivial task. Scala traits are closed under composition. Scala is supported by a sophisticated nominal type theory called vObj calculus [40].

8 AHEAD

AHEAD (Algebraic Hierarchical Equations for Application Design) is a feature modularization and composition technology based on step-wise development

[6][4][2]. It was created to address the issues of feature-based development of product-lines.

8.1 Feature Modules and Their Composition

AHEAD partitions features into two categories: *constants* that modularize any number of classes and interfaces, and *functions* that modularize classes, interfaces and their extensions.

AHEAD tools use a language, called *Jak* [4][5], that is a superset of Java. The implementation of constant features like `lp`, whose elements are standard classes and interfaces, uses pure Java constructs. To distinguish extensions of these elements, Jak provides modifier keyword *refines*. Also, to refer to the method being extended, Jak uses the construct `Super.methodName(args)`. For example, here is the Jak code of feature module `le`:

```
refines interface Exp { int eval(); }
refines class Lit implements Exp {
  public int eval() { return value; }
}
refines class Test {
  public void run() {
    Super.run();
    System.out.println( ltree.eval() );
  }
}
```

As described in Figure 3, this feature extends interface `Exp` with method `eval()`, extends class `Lit` with the corresponding implementation, and extends class `Test` by extending method `run()` with a call to `eval()` on `ltree`. `Super.run()` invokes the previously defined method `run()`. In the case of `LitAdd` it calls the `run()` method of `ap`. Constructor extensions follow a similar pattern, as illustrated in the following example, which extends the constructor of `Test` of feature `ap` by assigning variable `atree` a value:

```
refines Test() {
  Add atree = new Add( ltree, ltree );
}
```

The remaining feature modules are implemented in a similar way. Each feature is represented by a directory that contains files for each class and interface definition and extension. The command line to compose these directories to form `LitAdd` is:

```
composer -target=LitAdd lp ap le ae
```

8.2 Evaluation

Feature Definition. AHEAD can modularize all EPL program deltas into a cohesive unit. AHEAD provides tools to compile feature modules to bytecode and compose byte-code representations; however, this is not accomplished by separate compilation. Compilation uses global knowledge of all possible classes, interfaces, and members that can be present in a product-line [2].

Feature Composition. AHEAD feature modules are independent of the composition. The order in which features are composed is the order in which they are listed on the *composer* command line. AHEAD features are by definition closed under composition. A static typing model of feature modules for AHEAD is under development.

9 Perspective Beyond Individual Technologies

Let us step back from these implementation details to assess the fundamental nature of the problems that are being solved. We have seen that all five technologies can be used to implement EPL and how they satisfy, in different degrees, the properties required by feature modules. None of these technologies provide a satisfactory solution to the problem of building product lines, that is, they do not meet all the feature properties or express them in a verbose way. However, many common themes can be identified, even as each technology has particular strengths in meeting one or more of the properties.

In this section we show how the properties of feature definition and feature composition can be understood in terms of an algebra of program deltas. This simple algebra is an abstraction designed to express the underlying structure of feature modularization in product-line development. By hiding the details of particular technologies, this abstraction makes it easier to compare and contrast different technologies and suggests areas where the technologies could be improved or generalized. This discussion will, we hope, help encourage reliance on mathematically justifiable abstractions when developing new tool-specific or language-specific concepts [28].

A fundamental concept of metaprogramming is that programs are data and functions (a.k.a. *transforms*) map programs [7][42]. From this starting point, a program delta can be seen as a function that receives a program as input, adds something to it, and returns the extended program as output. Consider `Δrun()` of module `ap`. This delta adds a statement to method `run()` of class `Test` of the program received as input. Another example from `ap` is delta "`Add atree`", which adds member `atree` to class `Test`. For convenience, we refer to functions associated with program deltas by a single name. Thus we omit return types, parameters and their types in our function declarations. Using a mathematical notation, these two deltas are represented as:

```
Δrun(P)-> P′ where P′ is program with Δrun added to run() of Test of P
atree(Q)->Q′, where Q′ has field atree added to class Test of Q
```

When viewed in this way, a feature module like `lp` can be defined by:

```
lp = Test( Lit( Exp( Empty )))                                    (2)
```

where `Empty` is the empty program, and `Exp`, `Lit`, and `Test` are program deltas that add a new interface, and two new classes. To simplify notation further, we write expressions like this using the + operation, because it intuitively conveys the notion that we are building programs incrementally by adding program deltas. `(2)` now becomes:

```
lp = Test + Lit + Exp                                             (3)
```

where evaluation is from right to left. + denotes function composition; base terms are to the right and extensions are to the left. The choice of operator + was deliberately selected as (we will see) it exhibits composition properties that resemble those of elementary algebra. Next, we examine properties of this operator and relate them to the feature properties of Section 3.

Commutativity and Flexible Order. The order in which program deltas can be composed follows two simple rules. First, a program delta that references a data member or method must be composed *after* (to the left of) the delta that introduces that member or method. `(3)` is an example: `Exp` defines an interface, `Lit` adds a class that references this interface, and `Test` adds a class that references the class of `Lit`.

Second, program deltas that extend the same method are not commutative, because if their order is swapped, a different program will result. For example, changing the order in which `print` methods are added to method `run()` of class `Test` alters the output of a program. Summation is *commutative* (`A+B=B+A`) for arbitrary program deltas `A` and `B` if the first rule is not violated and `A` and `B` do not extend the same method. The evaluation property of *flexible order* relies on the non-commutativity property of operation +.

Substitution, Cohesion, and Closure. Module `ap` is defined by:

```
ap = △run + △Test + atree + Add                                    (4)
```

That is, (reading from right to left) it adds class `Add`, member `atree` to class `Test`, extends the `Test` constructor, and extends method run. When we compose `ap` with `lp`, we know the following equality holds because of *substitution* (i.e., replacing equals with equals):

```
ap + lp = (△run + △Test + atree + Add) + (Test + Lit + Exp)
```

That is, we know that the program produced by adding `ap` to `lp` must equal the sum of their deltas. *Cohesion* is the property that we can assign names `ap` and `lp` to summation expressions. *Closure* is the property that summation of deltas is itself a delta.[13]

Associativity and Flexible Composition. A common situation in product-line design is not only the addition of new features, but a refactoring of existing features into more primitive features.

Recall that in our EPL design, the `Print` operation is implemented in the `Print` column along with the declaration of the data types' fields and constructors. This design prevents, among other things, our ability to build programs without the `Print` operation. The solution is to refactor the `Print` column into two columns: `Print'` that implements operation `Print` exclusively, and `Core` that declares the data types with their fields and constructors. Figure 8 shows the refactoring of module `lp` into its core and non-core parts.

[13] Object-oriented classes contain methods that are mutually referential. One can factor each method into an empty (base) method and a program delta that adds the body. In this way, simple algebraic expressions can be written for mutually referential methods.

Class Test of module lp can be decomposed as:

Test = Δrun + run + Test^{C}+ ltree + Test^{S} where
Test^{S} is class Test { };
ltree is Lit ltree;
Test^{C} is Test() { ltree = new Lit(3); };
run is void run() { };
Δrun is ltree.print(); (5)

Superscript S stands for skeleton which is the declaration of the class without any members, and superscript C stands for constructor. Class Lit has a similar decomposition. Interface Exp can be decomposed as:

Exp = print^{I} + Exp^{S} where
Exp^{S} is interface Exp { };
print^{I} is void print(); (6)

Our refactoring lp in Figure 8 is captured by the following algebraic derivation:

1) lp = Test + Lit + Exp
2) lp =(Δrun + run + Test^{C} + ltree + Test^{S}) + (print + Lit^{C}+ value + Lit^{S}) + (print^{I} + Exp^{S})
3) lp = (Δrun + print + print^{I}) + (run + Test^{C} + ltree + Test^{S} + Lit^{C}+ value + Lit^{S} + Exp^{S})
4) lp = lp' + lp$_{Core}$

The first step recites (3). The second step substitutes the definitions of Test, Lit and Exp as in (5) and (6). The third step rearranges terms using the commutativity properties of summations. The last step uses an *associativity* property of summations (whose proof is simple)[14] and cohesion to express lp as a sum of lp' and lp$_{Core}$. Similar reasoning is applied to the other modules in the Print column to refactor them into a core and non-core part. The ability to refactor expressions is the property of *flexible composition*.

(a) lp'
```
// added to Exp
void print();

// added to Lit
void print() {
    System.out.print(value);
}

// added to run() of Test
ltree.print();
```

(b) lp$_{Core}$
```
interface Exp { }

class Lit implements Exp {
 int value;
 Lit (int v) { value = v; }
}

class Test {
  Lit ltree;
  Test() {
    ltree = new Lit(3);
   }
  void run() {  }
}
```

Fig. 8. Refactoring of lp to lp' + lp$_{Core}$

Compositional Reasoning, Static Typing, and Separate Compilation. Compositional reasoning is the ability to prove properties of a program from the properties of its components, which in our case are features, without reference to their

[14] + denotes function composition. Function composition is associative.

implementation [36]. By equating program deltas with functions (summations), we are relating *compositional reasoning* with *algebraic reasoning*. Doing so can be a substantial win for several reasons. First, an algebra provides a clean mathematical foundation for compositional reasoning and automation — both of which are needed in product-line development. Second, it changes our orientation on tool development and creation. Instead of inventing new tools with new abstractions and new conceptual models — e.g., the AspectJ, Hyper/J, Jiazzi, Scala, and AHEAD models are hardly similar and are difficult to compare — we have a single simple algebraic model that imposes clean abstractions *onto* tools, so that we can reason about programs in a tool-implementation-independent way.

Jiazzi provides an example of compositional reasoning: each feature module is statically typed. Jiazzi ensures that the composition of statically typed modules is itself a statically typed module. So not only does Jiazzi compose the *code* of individual features, it also computes (or verifies) an important property of a composition. Similar examples can be given from other technologies. All of this can be given an algebraic foundation. If we want property *p* of a summation, we need a composition operator +*p* (read *p*-sum) that tells us how to compose properties of constituent terms. So property *p* of module `lp`, denoted $\texttt{lp}_\texttt{p}$, is a *p*-sum of the *p* properties of its terms:

$$\texttt{lp}_\texttt{p} = \texttt{Test}_\texttt{p} +p\ \texttt{Lit}_\texttt{p} +p\ \texttt{Exp}_\texttt{p}$$

This idea (although not in an algebraic form) is common in the software architecture and product-line communities [46], and has been demonstrated elsewhere [6]. In the product-line and software architecture literature, feature modules map to *functional* requirements, and properties of modules and their compositions (such as the property of being statically typed) correspond to *non-functional* requirements.

The remaining property in our evaluation, *separate compilation*, is not a property of an algebraic model, but rather an engineering requirement of any implementation of the model.

10 Related Work

Relational query optimization is a classic example of the importance that algebra can play in program specification, construction, and optimization. SQL queries are translated to relational algebra expressions (i.e., compositions of relational algebra operators). A query optimizer rewrites the expression into semantically equivalent expressions where the goal is to minimize the expression (program) execution time. Readers will see that this is an example of compositional reasoning: the relational algebra expression defines the program, the optimizer composes a performance model of each operator to produce a performance model of that program [6].

The expression problem originated in the works of Reynolds [43] and Cook [15]. Torgersen [49] presents a concise summary of the research on this problem and four solutions that utilize Java generics. Though extensive, this literature focuses only on programming language design and separate compilation issues, and not about the requirements of feature modularity.

Masuhara et al. describe a framework to model the crosscutting mechanisms of AspectJ and Hyper/J [30]. Both are viewed as weavers parameterized by two input programs plus additional information such as where, what, and how new code is woven. Their focus is on the implementation of crosscutting semantics rather than on the broader software design implications that these mechanisms have.

Murphy et al. [37] present a limited study that uses AspectJ and Hyper/J to refactor features in two existing programs. The emphasis was on the effect on the program's structure and on the refactoring process, not in providing a general framework for comparison. Along the same lines, Driver [19] describes a re-implementation of a web-based information system that uses Hyper/J and AspectJ, but the evaluation is subjective and expressed in terms of factors such as extensibility, plugability, productivity, or complexity. Clarke et al. [13] describe how to map crosscutting software designs expressed as composition patterns (extended UML models) to AspectJ and Hyper/J, and evaluate their crosscutting capabilities to implement such patterns.

Coyler et al. [16] focus on refactoring tangled and scattered code into base code and aspects that could be considered as the features of a product line. They indicate that, based on their experience implementing middleware software, concerns (features) are usually a mixture of classes and aspects; a finding that corroborates the importance of feature cohesion.

For our evaluation we considered MultiJava, an extension of Java that supports symmetric multiple dispatch and modular open classes[11][12]. However, its focus is on solving the *augmenting method problem*, that consists on adding operations (methods) to existing type hierarchies. Given this constraint, it is not possible to implement EPL as it cannot add new fields, add new classes and interfaces, and extend existing methods and constructors. Similarly, *Classboxes* [8] are modules that provide method addition and method replacement (overriding without `super` reference). However, it is unclear if classboxes can support other program deltas such as adding new fields, or methods and constructor extensions.

The *Concern Manipulation Environment (CME)* [14] is a project that builds on the experience of Hyper/J and MDSoC. Among its goals is to provide support for the identification, encapsulation, extraction, and composition of concerns (features in this paper). CME architecture is geared towards supporting multiple modularization approaches. Thus it would be interesting to evaluate whether the software composition model we propose in this paper can benefit from the tool support that CME provides.

Mezini and Ostermann [35], present a comparison of variability management in product lines between *Feature-Oriented Programing (FOP)*, as in AHEAD, and Aspect-Oriented Programming, as in AspectJ. They identify as weaknesses in these technologies: a) features are purely hierarchical (extensions are made to some base code), b) support for reuse (extensions are tied to names not functionality), c) support for dynamic configuration (in FOP composition is static), and d) support for variability (aspects are either applied or not to an entire composition). They propose

Caesar[34] to address these issues. Caesar relies on Aspect Collaboration Interfaces, or ACIs, which are interface definitions for aspects (Caesar's aspects are similar to AspectJ's) whose purpose is to separate an aspect implementation from its binding. The association between these two is implemented with a *weavelet*, which must be deployed to activate advice either statically, when the object is created, or dynamically, when certain program block is executed. How these ideas could be applied to solve EPL is subject of an ongoing evaluation.

11 Conclusions and Future Work

Features express the kinds of variations product-line developers encounter in program development, because features represent increments in program functionality. Thus, it is natural to consider modularizing features as a way to modularize programs. Unfortunately, the code for features often cuts across classes, and thus traditional modularization schemes do not work well. New program modularization technologies have been proposed in recent years that have shown promise in supporting feature modularity. We have presented a classical problem in product-line design — called the Expressions Product-Line — to identify properties that feature modules should have. We have used these properties to compare and contrast five rather different technologies: AspectJ, Hyper/J, Jiazzi, Scala, and AHEAD. Our results showed that none of these technologies provide a satisfactory solution to the problem of building product-lines.

Instead of debating the merits of particular technologies, we focused on a topic that we believe has greater significance. Namely, product-line architects reason about programs in terms of their features, not in terms of their code or implementing technologies. We proposed an abstract model of features where compositional reasoning was related to algebraic reasoning. We showed how virtually all of the evaluation properties we identified in EPL were actually properties of an algebra. Namely: program deltas are functions that map programs, cohesion and closure under composition are associativity properties of function composition, flexible composition and flexible order is a consequence of the non-commutativity of certain functions, static typing is a property of a function (program delta) and is a property that can be predicted from an expression (i.e., a composition of deltas). Only the property of separate compilation dealt with engineering considerations of the algebra's implementation.

We believe the time has come for programming languages to play a more supportive role in product-lines and feature-based development. A consolidation of different modularization efforts is essential to this objective. We argued that such a consolidation should relate compositional reasoning with algebraic reasoning, because of its clean abstractions, the ability to automate compositional reasoning, and for giving an algebraic justification when adding new modularization concepts.

To continue this effort and because the full potential of the five technologies was not required, we foresee extending EPL and designing other case studies to help derive and illustrate further properties of feature modules (e.g. AOP quantification [28]). We are currently collaborating with proponents of other modularization technologies, such as Composition Filters [20], Caesar [34], and Framed Aspects [27], for this purpose.

Acknowledgments. We thank Sean McDirmid and Bin Xin for their help with Jiazzi, and Martin Odersky for his help with Scala. We are grateful to Axel Rauschmayer and Awais Rashid for their feedback on drafts of the paper, and the anonymous reviewers for their comments.

References

1. AspectJ. Programming Guide. `aspectj.org/doc/proguide`
2. AHEAD Tool Suite (ATS). `www.cs.utexas.edu/users/schwartz`
3. Batory, D., Geraci, B.J,: Composition Validation and Subjectivity in GenVoca Generators. IEEE Trans. Soft. Engr., February (1997) 67-82
4. Batory, D., Lopez-Herrejon, R.E., Martin, J.P.: Generating Product-Lines of Product-Families. Automated Software Engineering Conference (2002)
5. Batory, D., Liu, J., Sarvela, J.N.: Refinements and Multidimensional Separation of Concerns. ACM SIGSOFT, September (2003)
6. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Trans. Soft. Engr. June (2004)
7. Baxter, I.D.: Design Maintenance Systems. CACM, Vol. 55, No. 4 (1992) 73-89
8. Bergel, A., Ducasse, S., Wuyts, R.: Classboxes: A Minimal Module Model Supporting Local Rebinding. Joint Modular Languages Conferences JMLC (2003)
9. Beuche, D.:Composition and Construction of Embedded Software Families. Ph.D. Otto-von-Guericke-Universität Magdeburg (2003)
10. Bracha, G., Cook, W.: Mixin-based inheritance. OOPSLA (1990)
11. Clifton, C., Leavens, G.T., Millstein, T., Chambers, G.: MultiJava: Modular Open classes and Symmetric Multiple Dispatch for Java. OOPSLA (2000)
12. Clifton, C., Millstein, T., Leavens, G.T., Chambers, G.: MultiJava: Design Rationale, Compiler Implementation, and User Experience. TR #04-01, Iowa State University (2004)
13. Clarke, S., Walker, R.: Separating Crosscutting Concerns Across the Lifecycle: From Composition Patterns to AspectJ and Hyper/J. Technical Report UBC-CS-TR-2001-05, University of British Columbia, Canada (2001)
14. Concern Manipulation Environment (CME). `www.eclipse.org/cme/`
15. Cook, W.R.: Object-Oriented Programming versus Abstract Data Types. Workshop on FOOL, Lecture Notes in Computer Science, Vol. 173. Spring-Verlag, (1990) 151-178
16. Coyler, A., Rashid, A., Blair, G.: On the Separation of Concerns in Program Families. TRCOMP-001-2004, Computing Department, Lancaster University, UK (2004)
17. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
18. Douence, R., Fradet, P., Südholt, M.: Composition, reuse and interaction analysis of stateful aspects. AOSD (2004)
19. Driver, C.: Evaluation of Aspect-Oriented Software Development for Distributed Systems. Master's Thesis, University of Dublin, Ireland, September (2002)
20. Filman, R.E., Elrad, T., Clarke, S., Aksit, M.: Aspect-Oriented Software Development. Addison-Wesley (2004)

21. Flatt, M., Felleisen, M.: Units: Cool modules for HOT languages. PLDI (1998)
22. Findler, R.B., Flatt, M.: Modular Object-Oriented Programming with Units and Mixins. ICFP, (1998) 94-104
23. Jagadeesan, R., Jeffrey, A., Riely, J.: A Typed Calculus of Aspect Oriented Programs. Submitted for publication.
24. Kang, K., et al.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. CMU/SEI-90-TR-21, Carnegie Mellon Univ., Pittsburgh, PA, Nov. (1990)
25. Kiczales, G., Hilsdale, E., Hugunin, J., Kirsten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. ECOOP (2001)
26. Laddad, R.: AspectJ in Action. Practical Aspect-Oriented Programming. Manning (2003)
27. Loughran, N., Rashid, A., Zhang, W., Jarzabek, S.: Supporting Product Line Evolution with Framed Aspects. ACP4IS Workshop, AOSD (2004)
28. Lopez-Herrejon, R.E., Batory, D.: Improving Incremental Development in AspectJ by Bounding Quantification. SPLAT Workshop at AOSD (2005)
29. Lopez-Herrejon, R.E., Batory, D., Cook, W.: Evaluating Support for Features in Advanced Modularization Technologies. Extended Report. The University of Texas at Austin, Department of Computer Sciences, Technical Report TR-05-16, April (2005)
30. Masuhara, H., Kiczales, G.: Modeling Crosscuting Aspect-Oriented Mechanisms. ECOOP (2003)
31. McDirmid, S., Flatt, M., Hsieh, W.C.: Jiazzi: New age components for old-fashioned Java. OOPSLA (2001)
32. McDirmid, S., Hsieh, W.C.: Aspect-Oriented Programming with Jiazzi. AOSD (2003)
33. McDirmid, S., The Jiazzi Manual (2002)
34. Mezini, M., Ostermann, K.: Conquering Aspects with Caesar. AOSD (2003)
35. Mezini, M., Ostermann, K.: Variability Management with Feature-Oriented Programming and Aspects. SIGSOFT04/ FSE-12 (2004)
36. Misra, J.: A Discipline of Multiprogramming. Springer-Verlag (2001)
37. Murphy, G., Lai, A., Walker, R.J., Robillard, M.P.: Separating Features in Source Code: An Exploratory Study. ICSE (2001)
38. Odersky, M., et al.: An Overview of the Scala Programming Language. September (2004), `scala.epfl.ch`
39. Odersky, M., et al.: The Scala Language Specification. September (2004), `scala.epfl.ch`
40. Odersky, M., Cremet, V., Röckl, C., Zenger, M.: A nominal theory of objects with dependent types. ECOOP (2003)
41. Ossher, H., Tarr, P.: Multi-dimensional separation of concerns and the Hyperspace approach. In Software Architectures and Component Technology, Kluwer (2002)
42. Partsch, H., Steinbrüggen, R.: Program Transformation Systems. ACM Computing Surveys, September (1983)
43. Reynolds, J.C.: User-defined types and procedural data as complementary approaches to data abstraction. Theoretical Aspects of Object-Oriented Programming, MIT Press, (1994)
44. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.: Traits: Composable units of behavior. ECOOP (2003)
45. Schinz, M.: A Scala tutorial for Java programmers. September (2004), `scala.epfl.ch`
46. Software Engineering Institute. Predictable Assembly from Certified Components. `www.sei.cmu.edu/pacc`
47. Tarr, P., Ossher, H., Harrison, W., Sutton, S.M.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. ICSE (1999) 107-119
48. Tarr, P., Ossher, H.: Hyper/J User and Installation Manual. IBM Corporation (2001)
49. Torgersen, M.: The Expresion Problem Revisited. Four new solutions using generics. ECOOP (2004)

50. Wadler, P.: The expression problem. Posted on the Java Genericity mailing list (1998)
51. Walker, D., Zdancewic, S., Ligatti, J.: A Theory of Aspects. ICFP (2003)
52. Xin, B., McDirmid, S., Eide, E., Hsieh, W.C.: A comparison of Jiazzi and AspectJ. Technical Report TR UUCS-04-001, University of Utah (2004)
53. Zave, P.: FAQ Sheet on Feature Interaction. `www.research.att.com/~pamela/faq.html`
54. Zenger, M., Odersky, M.: Independently Extensible Solutions to the Expression Problem. Technical Report TR IC/2004/33, EPFL Switzerland (2004)

Separation of Concerns with Procedures, Annotations, Advice and Pointcuts

Gregor Kiczales and Mira Mezini

University of British Columbia, 201-2366 Main Mall,
Vancouver, BC V6R 1X4, Canada
gregork@acm.org
Technische Universität Darmstadt, Hochschulstrasse 10,
D-64289 Darmstadt, Germany
mezini@informatik.tu-darmstadt.de

Abstract. There are numerous mechanisms for separation of concerns at the source code level. Three mechanisms that are the focus of recent attention – metadata annotations, pointcuts and advice – can be modeled together with good old-fashioned procedures as providing different kinds of bindings: procedure calls bind program points to operations, annotations bind attributes to program points; pointcuts bind sets of points to various descriptions of those sets; named pointcuts bind attributes to sets of points; and advice bind the implementation of an operation to sets of points. This model clarifies how the mechanisms work together to separate concerns, and yields guidelines to help developers use the mechanisms in practice.

1 Introduction

Programming language designers have developed numerous mechanisms for separation of concerns (SOC) at the source code level, including procedures, object-oriented programming and many others. In this paper we focus on three mechanisms – metadata annotations [4], pointcuts [16] and advice [33] – that are currently attracting significant research [9, 10, 19, 34] and developer interest [1, 11, 12, 14, 20].[1]

Our goal is understand what kinds of concerns each mechanism best separates, and how the mechanisms work together to separate multiple concerns in a system. We also seek to provide developers with answers to questions about what mechanism to use in any given situation. To enable this, we study how the three newer mechanisms, along with good old-fashioned procedures, separate concerns in a simple example.

The study is focused on four key design concerns within the example. We present seven implementations of the example that use the mechanisms in different ways. We also present ten change tasks and how they are carried out in each implementation. Based on this, the paper provides:

1. An analysis of the degree to which the different mechanisms are able to separate and clarify the four design concerns in the seven implementations.

[1] The paper assumes a reading familiarity with pointcuts and advice as manifested by AspectJ [16] as well as the Java 1.5 metadata facility [4]. Metadata annotations, pointcuts and advice can appear in a wide range of other languages [3, 13, 21, 28, 31] [8, 30], but we do not explicitly discuss that generalization here.

2. An analysis of the degree of locality of each change task for each implementation, and a comparison of that locality to the static separation.
3. A unified model of the four mechanisms showing how they work together to separate concerns.
4. An initial set of guidelines for using the mechanisms in development practice.

The paper is structured as follows: Section 2 presents the example, its four key design concerns and the seven implementations. Section 3 analyzes the static locality of the concerns in each implementation, and the locality of the change tasks for each implementation. Section 4 presents the unified model of the mechanisms. Section 5 presents the usage guidelines. We finish with related and future work and a summary.

2 The Example

Our comparison of the mechanisms is based on seven implementations of a simple graphical shapes example [16, 18]. In this example, a number of graphical shapes are shown on a display. Each shape has its own display state, and when that state changes, the display must be signaled so it can refresh itself. This design is shown in Figure 1.

The key objects in the design are the shapes and the display. There is an abstract Shape class, with concrete Point and Line subclasses. (Assume there are other concrete shapes such as Triangle. To save space they are not discussed here.) There is a single Display class, and, for simplicity, there is just a single system-wide display.

2.1 Four Design Concerns

In addition to concerns involving the functionality of the shapes, the design comprises four key design concerns, which are shown as dotted line boxes in Figure 1.

Refresh-Implementation – What is the behavior and implementation of the actual refresh operation?

Context-to-Refresh – What context from the actual display state change points should be available to the refresh implementation?

When-to-Refresh – When should the display be refreshed?

What-Constitutes-Change – What operations change the state that affects how shapes look on the display, i.e. their position?

As is common, these concerns are interconnected. Our design resolves When-to-Refresh by deciding that refresh should happen immediately after each display state changes. This brings What-Constitutes-Change into focus as a concern that must be resolved. One could also argue the causality in the other direction, in that having thought about display state changes one then decides they should cause refreshes.

2.2 Seven Implementations

This rest of this section presents the code for seven implementations of the example. Discussion of the implementations is deferred until Section 3

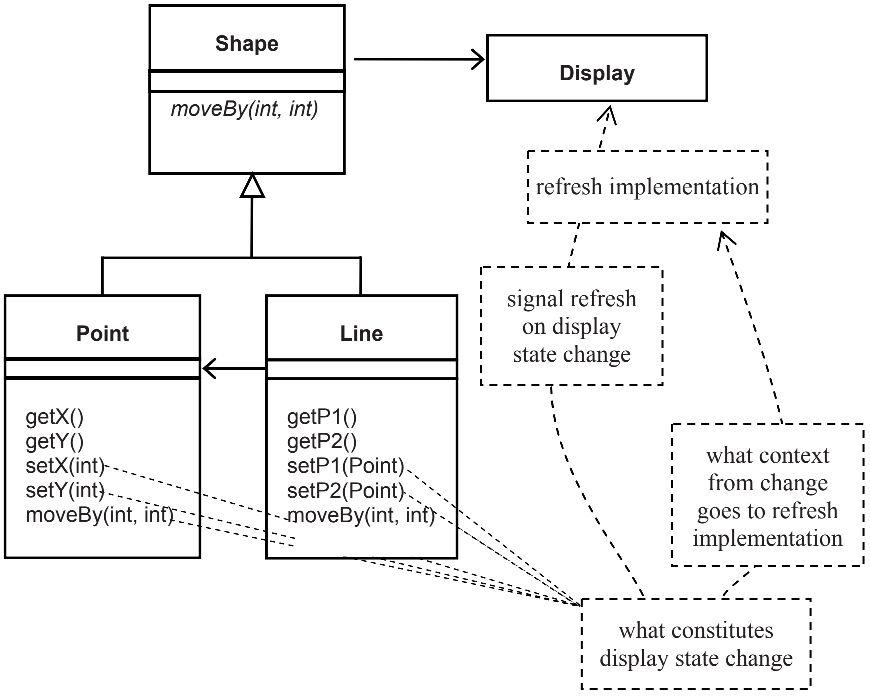


Fig. 1. The design of the graphical shapes program, showing the main classes and two additional design concerns not separated as classes

Straw-Man

The first implementation is a straw man, no good programmer would write this code today. Its purpose is to explicitly introduce procedures into the discussion.

In this implementation all of the methods that change display state directly include several lines of code that implement the actual display refresh. For example, the setX method looks like:

```
void setX(int nx) {
  x = nx;
  Graphics g = Display.getGraphics();
  g.clear();
  for( Shape s : Display.getShapes() ) {
    s.draw(g);
  }
}
```

GOFP

This implementation uses a good old-fashioned procedure (GOFP) to capture the refresh implementation. Each of the methods includes, at the end of the method, a call to a procedure (static method in Java) that refreshes the display.

```
void setX(int nx) {
  x = nx;
  Display.refresh();
}
```

The body of that procedure is the several lines of code that was duplicated in I1.

```
static void refresh() {
  Graphics g = getGraphics();
  g.clear();
  for( Shape s : getShapes() ) {
    s.draw(g);
  }
}
```

Annotation-Call
This implementation uses Java 1.5 metadata annotations [4]. Each method that changes display state has an annotation that says that executing the method should also refresh the display.

```
@RefreshDisplay
void setX(int nx) {
  x = nx;
}
```

A single after advice declaration serves to ensure that execution of methods with this tag calls Display.refresh(). The advice is written as:

```
after() returning: execution(@RefreshDisplay * *(..)) {
  Display.refresh();
}
```

There are other ways to associate run-time behavior with annotations, typically involving ad-hoc post-processors. We use advice in this paper because it is simple and compatible with the rest of the paper.

Annotation-Property
This implementation differs from the previous one only in the name of the annotation. Here the annotation name describes a property of the method – that it changes state that affects the display of the shape – rather than directly saying that executing the method should refresh the display. So the methods look like

```
@DisplayStateChange
void setX(int nx) {
  x = nx;
}
```

Again, a separate advice declaration says that execution of methods with the DisplayStateChange annotation should call Display.refresh().

```
after() returning: execution(@DisplayStateChange * *(..)) {
  Display.refresh();
}
```

Anonymous-Enumeration-Pointcut

This implementation uses an anonymous enumeration-based pointcut to identify method executions that change display state. So the methods have no explicit marking, and simply look like:[2]

```
void setX(int nx) {
  x = nx;
}
```

The entire implementation of signaling a display refresh consists of a single advice on an anonymous pointcut that explicitly enumerates the six methods; the body of the advice calls Display.refresh().

```
  after() returning: execution(void Shape.moveBy(int, int)
                   || execution(void Point.setX(int))
                   || execution(void Point.setY(int))
                   || execution(void Line.setP1(Point))
                   || execution(void Point.setP2(Point)) {
  Display.refresh();
}
```

Named-Enumeration-Pointcut

In this implementation the pointcut from the previous implementation is pulled out and given an explicit name. Again, the method bodies require no marking to enable display refresh signaling.

```
void setX(int nx) {
  x = nx;
}
```

The pointcut and advice are:

```
pointcut displayStateChange():
  execution(void Shape.moveBy(int, int)
   || execution(void Point.setX(int))
   || execution(void Point.setY(int))
   || execution(void Line.setP1(Point))
   || execution(void Point.setP2(Point));

after() returning: displayStateChange() {
  Display.refresh();
}
```

Named-Pattern-Pointcut

In this implementation only the pointcut differs from the previous implementation. Rather than enumerating the signatures of the methods that change display state, this implementation relies on the naming convention the methods follow to write a more concise pointcut. Again, the method bodies require no marking to enable display refresh signaling.

[2] Even though the method is not explicitly marked by the programmer, IDE support such as the ADJT Eclipse plug-in will show that the advice exists, for example with a gutter marker next to the method declaration [27].

```
void setX(int nx) {
  x = nx;
}
```

The pointcut and advice are:

```
pointcut displayStateChange():
  execution(void Shape.moveBy(int, int)
  || execution(void Shape+.set*(..));

after() returning: displayStateChange() {
    Display.refresh();
}
```

The `execution(void Shape+.set*(..))` pointcut means execution of any method defined in Shape or a subclass of Shape, that returns void, has a name beginning with ‘set’, and takes any number of arguments.

3 Analysis of the Implementations

Our analysis of the different mechanisms is based on assessing the degree to which the seven implementations separate the four design concerns identified in Section 2.1. The assessment uses three criteria: locality of implementation, degree to which the implementation is explicit rather than implicit, and locality of change in a simple evolution experiment. The assessment of locality and explicit implementation is discussed in Section 3.1. The locality of change assessment is covered in Section 3.2. All three assessments are summarized in Table 1.

3.1 Locality and Explicit Representation

One way to compare how the implementations separate these design decisions is whether the code that implements the decision is localized. Another criterion is the degree to which the implementation of the decision is captured explicitly as opposed to implicitly. This analysis is summarized in the top part of Table 1.

The capture of Refresh-Implementation is implicit and non-localized in Straw-Man. There is no single place in the code that explicitly says that display refresh is implemented by the several lines of code. Instead, each method that the developer decided constitutes a display state change includes code that implements refresh. In the GOFP and subsequent implementations, the refresh procedure declaration captures this concern in an explicit and localized way. The declaration is read as saying “this is the refresh implementation – bind Display.refresh() to this code”.

The capture of Context-to-Refresh is implicit and non-localized in Straw-Man. No single place in the code explicitly says that no values from the change context are available to the display refresh implementation. In GOFP, the procedure declaration and every call to the procedure explicitly say that no arguments are passed, so this concern is explicit. But because this is expressed in the procedure and all the calls to it, it is non-localized. In the Annotation implementations there is a single call to the procedure, so this concern is captured explicitly and in two places. The same is true for the

Anonymous-Enumeration-Pointcut implementation. In the last two implementations the named pointcut also expresses this concern, so it is captured explicitly in three places.

The capture of the When-to-Refresh is implicit and non-localized in Straw-Man, GOFP and Annotation-Call. It is localized but implicit in Anonymous-Enumeration-Pointcut. No single place in these implementations explicitly says that execution of methods that change display state should cause a display refresh. In GOFP the scattered calls to Display.refresh() are implicitly about the fact that the affected methods change display state and so must refresh; but all they say explicitly is that the affected methods call Display.refresh(). The same is true for the scattered RefreshDisplay tags in Annotation-Call. In Anonymous-Enumeration-Pointcut, the pointcut localizes the description of what constitutes change, but because no name is given to it, the binding of when to refresh is not to a clear notion of on display state changes, but instead to an enumerated set of conditions. In the other implementations, this concern is explicit and localized in the after advice declarations, which say that any display state change should cause a refresh.

The capture of the What-Constitutes-Change is implicit and non-localized in Straw-Man, GOFP and Annotation-Call – no single place in these implementations explicitly says that execution of the four setter methods and the two moveBy methods changes display state. In Annotation-Property, the DisplayStateChange annotations capture this concern in an explicit, but non-localized way. In Anonymous-Enumeration-Pointcut, this concern is localized, but implicit. In the two named pointcut implementations this concern is localized and explicit. The Named-Pattern-Pointcut captures the decision about what methods change display state, as well as a rule for what methods are considered to change display state. The variation among the pointcut based implementations is discussed in more depth in Section 5.

Names Matter

The two annotation-based implementations differ only in the name of the annotation, but come out significantly different in our separation of concern analysis. Annotation-Call has the same properties as GOFP with regard to When-to-Refresh and What-Constitutes-Change. This should not be surprising since in Annotation-Call the annotation name makes it feel like alternate syntax for a procedure call, or a syntactic macro [6, 7]. So, like GOFP, Annotation-Call, is conflating these two concerns and simply saying to call refresh at certain points.

On the other hand, in Annotation-Property, When-to-Refresh is captured explicitly and in just one place in the code; What-Constitutes-Change is captured explicitly but is not localized. The different annotation name causes both concerns to be explicit. That names matter is not surprising to programmers, but it is important to note its significance in this case. We return to this issue in Section 5.

3.2 Ease of Evolution

This section analyzes the implementations in terms of how well they fare when performing a set of ten representative change tasks. Most tasks affect just a single concern, reflecting a good modularity in the concern model itself. The question we explore now is what must be done to the code to perform each task – how many edits and how localized are they. The analysis is summarized in the lower part of Table 1 by showing, for each change and each implementation, how many places in each implementation have to be visited and possibly edited by the programmer.

Double-buffering – changes the refresh implementation to use double buffering. So it is a change to just the Refresh-Implementation concern. In Straw-Man, the programmer must edit the refresh implementation code that appears in all the display state change methods. For GOFP and all other implementations only the Display.refresh() procedure must be edited. In Table 1, the Double-Buffering row shows 'n' in the first column and 1 in the remaining columns. This is one of the reasons we have learned to introduce a procedure in such cases.

Pass-Changed-Object – provides the actual shape that has changed to the refresh implementation, so that it can optimize refresh based on that information. This constitutes a change to both Refresh-Implementation and Context-to-Refresh. In Straw-Man, this change task involves editing all the state change methods. In GOFP it involves editing the procedure declaration and the call sites in all the state change methods. In the remaining implementations this involves editing the procedure, advice and pointcut declarations. The procedure is edited to accept the shape as an argument, the call sites are edited to pass the current object, and the pointcuts are edited to make the current object accessible.

Disable-Refresh – simply disables activation of display refresh when the state of shapes changes. So this is a change to just When-to-Refresh. In Straw-Man this change requires editing all the state change methods to delete the refresh implementation. GOFP and Annotation-Call require editing all the methods to remove the call to the refresh procedure or the refresh annotation respectively. In the last four implementations this change can be accomplished by removing the aspect containing the advice from the system, or by editing the aspect to delete the advice if for some reason the aspect should remain. The Disable-Refresh table row shows 'n' in the first 3 columns and '1' in the last four.

One might argue that GOFP and Annotation-Call can accommodate Disable-Refresh more expeditiously – for GOFP, one could simply "comment out" the body of the refresh procedure declaration, and for Annotation-Call one could delete the advice declaration. But these alternatives are problematic. There may be other callers of the refresh procedure (or clients of the tag), since nothing has marked the procedure or the tag as particular to handling this kind of refresh activation. Even if there are no other callers, the expeditious changes make the code confusing – the reader sees a call to refresh (or the annotation), but must learn elsewhere that they do not do anything.

A programmer might deal with this by introducing an additional procedure, perhaps called Shape.fireDisplayStateChange(), and have that procedure call Display.refresh(). Then this change can be easily accommodated by making the body of the new procedure empty. This has the same effect of introducing the intermediate annotation, and has the same separation properties as Annotation-Property. Other more elaborate rendezvous mechanisms could be used as well. Having this extra procedure vs. not having it is similar to the difference between the two annotation-based implementations.

Reuse-What-Constitutes-Change adds logging of display state changes. So it reuses What-Constitutes-Change, but does not actually change any of the design concerns. In Straw-Man all the state change methods are edited to add logging code. In GOFP all the state change methods are edited to add a call to a logging operation. In Annotation-Call, each method gets an annotation and a new advice is defined. In the last four implementations, a new advice is defined; in Annotation-Property it references the @DisplayStateChange annotation, in the anonymous pointcut it duplicates the anonymous enumeration-based pointcut, and in the named pointcut implementations it references the displayStateChange pointcut. For all but Straw-Man the table includes an extra count assuming the logging operation must be defined as a procedure.

Again, one might argue that this can be accomplished more expeditiously in GOFP and Annotation-Call, simply by directly editing the refresh procedure or the advice to do the logging. This however, associates the logging with the activation of the refresh, rather than directly with the state changes.

Refresh-Top-Level-Changes-Only ensures that in recursive state change methods (e.g. moveBy on Line calls moveBy on Point, which calls setX and setY on Point) only the top-level display state change method causes a refresh. This prevents multiple refreshes for such methods. So it is a change to the When-to-Refresh concern. In Straw-Man and GOFP this change requires editing all the state change methods, to introduce some mechanism that can detect recursive state change method calls and prevent the sub-calls from calling refresh. A common pattern for doing this is to add a second parameter to all the state change methods, indicating whether they are part of a recursive call. Often a second overloaded method is introduced to handle this. In Java the programmer can use thread local state to do this in a more elegant way.

In the implementations that use pointcuts (all after GOFP), this can be done by editing the pointcut to use the cflowbelow primitive to filter out recursive calls; in the named pointcut implementations the AspectJ code for this would involve modifying the advice to be:

```
after() returning: displayStateChange()
                   && !cflowbelow(displayStateChange()) {
    Display.refresh();
}
```

which is read as saying to call refresh after any display state change that is not itself within the control flow of another display state change.

The next five changes all affect What-Constitutes-Change in different ways.

Add-Related-Class adds a new Circle subclass of Shape. The new class has setX, setY, setRadius and moveBy methods that constitute display state changes. This represents a modification of the What-Constitutes-Change concern. Straw-Man, GOFP and both annotation-based implementations each require that all the new state change methods be appropriately edited. The two enumeration-based pointcut implementations require that the pointcut be edited. The pattern-based pointcut does not need to be edited, but it must be at least examined to ensure that the new methods are covered by the pointcut.

The next two changes have the same implications for all implementations as Add-Related-Class. They are included nonetheless because they are typical changes to expect in such a system.

Add-Related-Method adds a new Line.setColor(Color) method that should be considered to change display state.

Table 1. Analysis of the seven implementations. The top part of the table shows many places in the code implement the concern, and whether the implementation is **E**xplicit or **I**mplicit; ‘n’ means each of the display state change methods. The bottom part of the table summarizes the change task analysis, showing the number of places each implementation must be edited for each change. The ‘n’ notation indicates that the number goes up as the number of shape classes increases, whereas other numbers are constant. The ‘*’ indicates that the code is only examined, not edited. In this part of the table the first column shows what concerns each tasks changes

Implementations		Straw-Man	GOFP	Annotation-Call	Annotation-Property	Anonymous-Enumeration-Ptc.	Named-Enumeration-Ptc.	Named-Pattern-Ptc.
Design Concerns								
Refresh-Implementation		I, n	E, 1					
Context-to-Refresh		I, n	E, n+1	E, 2 / 3				
When-to-Refresh		I, n			E, 1	I, 1	E, 1	
What-Constitutes-Change		I, n			E, n	I, 1	E, 1	
Change Tasks	*Concerns*							
Double-Buffering	RI	n	1					
Pass-Changed-Object	RI, CtR	n	n+1	2			3	
Disable-Refresh	WtR	n			1			
Reuse-What-Constitutes-Change	WCC	n	n+1	n+2	2			
Refresh-Only-Top-Level-	WtR	n		1		1 + 1		
Add-Related-Class	WCC	each new method				1		1^*
Add-Related-Method	WCC	each new method						
Rename-Methods	WCC	0						
Add-Unrelated-Class	WCC	0						
Add-Unrelated-Method	WCC	0						1

Rename-Methods renames the Line.setP1(Point) and Line.setP2(Point) methods to Line.setEnd1(Point) and Line.setEnd(2).

Add-Unrelated-Class adds an entirely unrelated class to the system. It does not change any of the four concerns. None of the implementations require any editing or examination to perform this change.

Add-Unrelated-Method adds a new Shape.setOwner(Owner) method that has nothing at all to do with display state. This change also does not change any of the four concerns. The first six implementations require no editing, but the pattern-based pointcut must be edited to exclude the new setOwner method.

4 Uniform Characterization of Mechanisms

The above analysis suggests that one useful way to characterize the four mechanisms is as establishing different kinds of bindings along a path from points in a program to the implementation of an operation that must execute at those points. As shown in Figure 2, each mechanism introduces an explicit intermediate step along the path, and makes an explicit binding between those steps. These explicit steps and bindings work together to separate larger, higher-level concerns such as the four discussed here.

In these terms, a procedure call binds a point in the program to an operation – it says call this operation at this point in the program execution. A procedure declaration binds the operation to an implementation. So the effect of using a procedure – a declaration and one or more calls to it – is to introduce an explicit operation (the procedure), bindings from points in the program to the operation (calls), and a binding from the operation to the implementation (the declaration).[3] Annotations, pointcuts and advice introduce other explicit intermediate elements and bindings.

In discussing the relation between annotations and pointcuts, we use the following terminology: *Annotations* are the syntactic identifiers described by JSR-175 [4] that the programmer places in the program (i.e. @DisplayStateChange). Properties are the characteristics of points on which pointcuts can match, including class and method names, access modifiers etc. *Pointcut names* are the programmer defined names for pointcuts. We use the term *attribute* to include both annotations and pointcut names. In other words, attributes are user-defined names that can be attached to program points.

Annotations bind attributes to program points. An annotation such as @DisplayStateChange binds the DisplayStateChange attribute to the program point.

There are several different kinds of pointcuts. Enumeration-based pointcuts make a set of points explicit, and establish a binding between the set and each of the points.

Pattern based pointcuts make a set of points and the fact that they conform to a common pattern explicit; they also establish a binding between the set and the points. Property-based pointcuts, such as 'execution(public com.acme.*.*(..))' do the same for properties instead of patterns. Annotation-based pointcuts do this for annotations.

Named pointcut declarations establish a binding between an attribute (the pointcut name) and a (possibly singleton) set of points.

[3] We use the term *procedure declaration* to refer to a construct that defines both signature and implementation, such as a static method declaration within a class in Java, as opposed to a construct that just declares the procedure's signature.

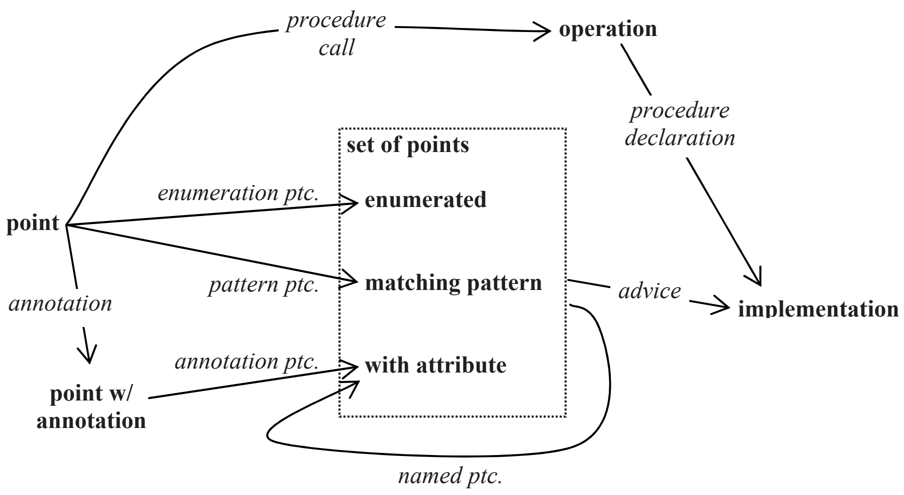


Fig. 2. Intermediate elements and bindings established by the mechanisms. Elements are shown in boldface, the mechanisms are in italics

Advice can be used with any kind of pointcut to bind between the intermediate step that pointcut makes explicit and the implementation of an operation to execute at those points.

This characterization provides an interesting perspective on one difference between AspectJ and AspectWerkz [5]. In AspectJ, the body of an advice is a code block. But in AspectWerkz, advice has no code block; instead it is written as a method, with an annotation that contains the kind of advice and the pointcut.[4] In terms of our model, this means that in AspectWerkz, the advice construct binds to an operation, whereas in AspectJ it binds to an operation implementation. So AspectWerkz provides an extra binding step. In AspectJ the programmer can achieve the extra binding step simply by having the advice body call a procedure.

5 Usage Guidelines

Our model of how the different mechanisms serve to separate concerns suggests a way to approach the process of deciding which mechanism(s) to use in a given situation. The following guidelines are organized around the binding steps in Figure 2 and work to help the programmer decide which path through the figure is most appropriate in a given situation. For each guideline, we discuss how it is validated from by the study described above.

Procedures
If an operation is needed at a given point, then using a procedure (call and declaration) serves to make the operation explicit and local, and to make the binding

[4] AspectJ 5 includes both alternatives.

from the point to the operation explicit. This can improve comprehensibility of both the operation and the context, enable reuse of the operation in other contexts, and facilitate later change to the operation.

Comparing Straw-Man to the subsequent implementations, we see that the use of a procedure makes Refresh-Implementation explicit and local. Separating this concern explicitly makes its implementation more clear, and also clarifies the contexts where the operation is invoked (e.g. the setX method). The refresh procedure can easily be called from other points (reused). When Refresh-Implementation changes in the Double-Buffering and Pass-Changed-Object tasks the implementations that use the procedure fare better. None of this is a surprise; we are all familiar with these properties of using procedures. We are elaborating this here only to show how this set of guidelines encompasses the familiar case of procedures and to lay a foundation for discussion guidelines regarding annotations, pointcuts and advice.

Advice and Pointcuts

If an operation is needed at a given set of points then using advice and pointcuts serves to make the binding from the set to the operation explicit and local, which can improve comprehensibility and evolvability in some cases. In particular, consider using advice and pointcut rather than multiple procedure calls if: (i) more than a small number of points must invoke the operation, (ii) the binding between the points and the operation may be disabled or otherwise be context-sensitive, or, (iii) the calling protocol to the operation may change.

All the implementations that use advice and pointcuts (Annotation-Property and on) make the calling protocol to Display.refresh explicit and localized. So they support part iii of this guideline.

But in this regard it is worth looking carefully at the way the implementations that use advice and pointcuts enhance the capture of When-to-Refresh (WtR) and What-Constitutes-Change (WCC). Annotation-Call does not improve WtR or WCC over GOFP. Annotation-Property makes WtR explicit and local and makes WCC explicit but non-local. With Anonymous-Enumeration-Pointcut both concerns are local, but are once again implicit. In the named pointcut implementations both concerns are local and back to being explicit. Since all these implementations use advice and pointcuts of some form, this suggests an interaction between using advice and the form of the pointcut used in the advice, which leads to the next guideline.

Attributes – Named Pointcuts or Annotations

If a set of points used in an advice has a common attribute, then using a named pointcut or an annotation can make that common attribute explicit. Using named pointcuts makes the attribute explicit and local, annotations make it explicit and non-local. When using named attributes, choose a name that describes what is true about the points, rather than describing what a particular advice will do at those points.

This guideline is supported by the Annotation-Property and the two named pointcut implementations. What-Constitutes-Change is made explicit in all three of these implementations. It is made local in the two named pointcut implementations. In each case, the capture of When-to-Refresh also benefits, which is the link to the previous guideline.

As with procedures, the motivation to make the additional bindings and intermediate steps explicit using advice and named attributes comes from comprehensibility, reuse, evolution and other considerations. Comprehensibility is subjective, but to our eye, Annotation-Property and the two named pointcut implementations are the easiest to understand because they make all the steps leading up to a refresh clear. They clearly say "there is an explicit concept of display state change", "here are points that constitute such changes"; and "call refresh at those points". Straw-Man, GOFP and Annotation-Call make it clear that refresh is happening, but not why. Anonymous-Enumeration-Pointcut makes it clear that there is a general condition that causes refresh to happen, but without a pointcut name the abstraction of the condition is not clear.

In terms of reusability, because Annotation-Property and the two named pointcut implementations make the (d/D)isplayStateChange attribute explicit, they make it easy to reuse What-Constitutes-Change in the change task.

In terms of evolution, making the binding from the (d/D)isplayStateChange attribute to the refresh signaling behavior explicit makes the Disable-Refresh change task easy.

The Annotation-Call and Annotation-Property implementations demonstrate the importance of choosing good annotation names. In Annotation-Call the name of the annotation is such that it fails to introduce the intermediate step and make clear why refresh is happening. A named pointcut with a similar name would have similar problems.

Introducing additional attribute names does not always add value. When writing procedural code, most programmers are unlikely to define a new onePlus procedure for the expression 'x + 1'. They could, but in this case the primitive expression is sufficiently clear that it is usually left in line. Named abstraction has to stop at some point, or else programs would never reach primitives.

The same is true for attributes. The pointcut 'execution(public com.acme.*.*(..))' is sufficiently clear that it usually does not warrant a named pointcut. On the other hand 'execution(* Shape+.set*(..))' probably does warrant the displayStateChange named pointcut.

Enumeration, Property, Pattern-Based Pointcuts and Annotations

The previous guidelines leave open the question of what mechanism to use to establish the binding between the individual point(s) and the actual set of points. The choices are enumeration-based pointcuts, name-pattern based pointcuts, property-based pointcuts or annotations.

Prefer enumeration-based pointcuts when: (i) it is difficult to write a stable property-based pointcut to capture the members and (ii) the set of points is relatively small.

Prefer property- or pattern-based pointcuts when: (i) it is possible to write one that is stable or (ii) the set of points is relatively large (more than ten).

Use annotations to mark points when three things are true: (i) it is difficult to write a stable property-based pointcut to capture the points, (ii) the name of the annotation is unlikely to change, and (iii) the meaning of the annotation is an inherent to the points, rather than a context-dependent aspect of the points only true in some configurations.

In addition, lean towards annotations when the property that defines inclusion in the set is an inherent property of the points, and lean towards other pointcuts when the binding from points to the set might change non-locally, or come into existence non-locally.

The implementations after GOFP provide some support for these guidelines, but the example is too small to fully support them.

The difference between how Named-Enumeration-Pointcut and Named-Pattern-Pointcut fare for Add-Unrelated-Method both shows the concern about pattern-based pointcuts, and also shows that using stable patterns can mitigate that concern.[5] For example ‘Shape+.set*(*)’ means methods defined on Shape or a subtype of Shape, for which the name begins with set, and that have a single argument. This pattern has good stability both because it is restricted to a small part of the type hierarchy, and because it is based on a well-established Java naming convention. By contrast, ‘set*(..)’ is less stable, it covers any type of object, and methods with any number of arguments.

Once again, the difference between Annotation-Call and Annotation-Property supports the importance of annotation names. As formulated above, the guideline is intended to reduce the likelihood that the name will need to change, and will make it more natural to reference the same annotation in other aspects or in compositions of pointcuts based on the annotation. For example DisplayStateChange may be reasonable as an annotation. But MakesRemoteCall may not be, because it may depend on a particular deployment configuration rather than always being true of a method.

While the guidelines for preferring property and pattern-based pointcuts when the number of points is large and it is possible to write such pointcuts are not supported by this study, they seem fairly straightforward, although it would be valuable to validate them, and all the other above guidelines, in a larger case study.

6 Related Work

There have been a number of characterizations of aspect-oriented programming (AOP) mechanisms: as a means for modularizing crosscutting concerns [16, 17], in terms of obliviousness and quantification [10], in terms of a common join point model framework [25] and others. By contrast, the focus of this paper is on analyzing the separation of concern properties of annotations, pointcuts and advice, and describing those as binding mechanisms similar to procedures.

The work described in [10] and [24] is closer to this paper in that they characterize AOP mechanisms as a new step in “introducing non-locality in our programs” [10], specifically as a means of binding points in the execution space [24]. But, they do not consider annotations. They also do not focus on the way in which the mechanisms compare for separating different kinds of concern or provide guidelines for choosing among the mechanisms.

The discussion by Lopes et al. [22] shares with this paper the view that pointcuts act as a kind of referencing mechanism. The focus in [22] is more on motivating and

[5] Practicing AspectJ developer report that the restrictions that come from the use of name patterns often benefits their code. The patterns force them to regularize the rules they use for naming, and that helps with overall system comprehensibility. Nonetheless, this issue is motivating a variety of important research in more powerful pointcut languages, that make it possible to express pointcuts in terms of properties that are more accurate and robust than name patterns [24, 35].

speculating about future "more naturalistic" referencing mechanisms that go beyond current pointcut mechanisms. On the contrary, our focus is on characterizing and assessing state-of-the-art mainstream pointcut mechanisms and especially on providing guidelines for using them.

Rinard et al. [29] propose a classification and an analysis system for AOP programs that classifies interactions between aspects and methods to identify potentially problematic interactions (e.g., caused by the aspect and the method both writing the same field), and guide the developer's attention to the causes of such interactions. Hence, their focus is different than ours. They also do not discuss annotations, and only indirectly suggest usage guidelines. To the extent they do suggest guidelines there appear to be no conflicts between their work and ours.

Baldwin and Clark have developed a general framework for assessing the value of modularity in technical systems [2]. Sullivan et al. [32] show how this framework can be applied to software systems. Lopes and Bajracharya [23] went on to apply the framework to AOP systems. The Baldwin and Clark framework is more heavy-weight than ours, and seems more suitable for architectural decision making than what we discuss here. But again, there does not appear to be any inherent conflict between the analyses. One interesting next experiment would be to see how the guidelines we develop interact with the analyses and net option value framework used by these researchers.

Our guidelines are 'bottom-up' or in-situ in nature. They are focused on how a developer makes isolated decisions about what mechanism to use guided by design goals. By contrast, Jacobsen and Ng have proposed a methodology for designing systems in an aspect-oriented style [15]. Again, there appears to be no contradictions between our guidelines and their methodology.

The work presented in [26] also involves an assessment of pointcut mechanisms with respect to how well programs using them fare in presence of change, as compared to equivalent OO programs that use method calls only. That assessment does not consider annotations, and is primarily on assessing the need for pointcut mechanisms that refer to more dynamic properties of join points than possible today. The design and implementation of such pointcuts is the main focus of their paper.

7 Future Work

The analysis and guidelines in this paper are based on first-principles analysis with a single small example. Based on this, there are several attractive avenues for future work.

One next step would be large-scale validation of these guidelines. There are (at least) two dimensions of improvement. First, they could be validated against a larger sample of code developed by experts. While attractive, at present there do not appear to be large bodies of suitable open source code to work with, although this appears to be changing rapidly.

A second line of work would be to validate these guidelines in some form of user study in which programmers are asked to work with the guidelines in a controlled experiment.

As discussed in Section 6, it would also be interesting to develop a detailed account of how the guidelines we propose interact with classifications such as in [29], architectural analyses such as in [2], and design methodologies such as in [15].

8 Summary

Metadata annotations, pointcuts and advice are useful techniques for separating concerns in source code. To better understand and be able to work with these mechanisms, we propose a characterization in which each is seen as making a different kind of binding: annotations bind attributes to program points; pointcuts create bindings between sets of points and descriptions of those sets; named pointcuts bind attributes to sets of points; and advice bind the implementation of an operation to sets of points.

This characterization yields insight into how the mechanisms relate and suggests areas for improvement. It also yields guidelines for how to choose among the mechanisms in the course of programming with them. The guidelines can be phrased in terms of deciding which kind of binding is appropriate in a given situation or they can be formulated in more prescriptive terms that may be more appropriate in some contexts.

The model and guidelines proposed here provide a good basis for further research and near-term development. We expect improvements to the model and guidelines as the combined use of annotations, pointcuts and advice grows.

Acknowledgements

The author would like to thank Mik Kersten, Mira Mezini and Gail Murphy for fruitful discussions on the ideas developed in this paper. Thanks also go to Andrew Eisenberg, Mik Kersten, Gail Murphy, Kevin Sullivan and Maria Tkatchenko who provided detailed comments on earlier drafts of the paper.

This work is partially funded by the Natural Sciences and Engineering Research Council of Canada (NSERC), IBM Canada Ltd. and the European Network of Excellence in Aspect-Oriented Software Development (AOSD-Europe).

References

1. The Server Side Symposium: AOP Expert Panel, 2004, http://www.theserverside.com/news/thread.tss?thread_id=30564.
2. Baldwin, C.Y. and Clark, K.B. Design Rules: The Power of Modularity. MIT Press, 2000.
3. Bergmans, L. and Aksit, M. Principles and Design Rationale of Composition Filters. in Filman, R.E., Elrad, T., Aksit, M. and Clarke, S. eds. Aspect-Oriented Software Development, Addison Wesley Professional, 2004, 63 - 95.
4. Bloch, J. A Metadata Facility for the Java Programming Language, 2004.
5. Boner, J., AspectWerkz http://aspectwerkz.codehaus.org/.
6. Bryant, A., Catton, A., Volder, K.D. and Murphy, G.C., Explicit programming. Aspect-Oriented Software Development, 2002, ACM Press, 10-18.

7. Cheatham, T.E., JR., The introduction of definitional facilities into higher level programming languages. (AFIPS) Fall Joint Computer Conference, 1966, Spartan Books, 623-673.
8. Coady, Y., Kiczales, G., Feeley, M. and Smolyn, G., Using AspectC to improve the modularity of path-specific customization in operating system code. Foundations of Software Engineering (FSE), 2001, ACM Press, 88 - 98.
9. Elrad, T., Aksit, M., Kiczales, G., Lieberherr, K. and Ossher, H. Discussing aspects of AOP. COMMUNICATIONS OF THE ACM, 44 (10). 33-38.
10. Filman, R.E., Elrad, T., Aksit, M. and Clarke, S. (eds.). Aspect-Oriented Software Development. Addison Wesley Professional, 2004.
11. Gradecki, J. and Lesiecki, N. Mastering AspectJ: Aspect-oriented Programming in Java. Wiley, Indianapolis, Ind., 2003.
12. Group, G., Hype Cycle for Application Development, 2004, http://www4.gartner.com/DisplayDocument?doc_cd=120914.
13. Hirschfeld, R. AspectS - Aspect-oriented programming with squeak. Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World, 2591. 216-232.
14. Jacobson, I. and Ng, P.-W. Aspect-Oriented Software Development with Use Cases. Addison-Wesley, 2003.
15. Jacobson, I. and Ng, P.-W. Aspect-Oriented Software Development with Use Cases. Addison Wesley Professional, 2004.
16. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G., An Overview of AspectJ. European Conference on Object-Oriented Programming (ECOOP), 2001, Springer, 327-355.
17. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. and Irwin, J., Aspect-oriented programming. European Conference on Object-Oriented Programming (ECOOP), 1997, 220-242.
18. Kiczales, G. and Mezini, M., Aspect-Oriented Programming and Modular Reasoning. ACM International Conference on Software Engineering, 2005 (to appear).
19. Krishnamurthi, S., Fisler, K. and Greenberg, M. Verifying aspect advice modularly. Foundations of Software Engineering (FSE). 137 - 146.
20. Laddad, R. AspectJ in action: practical aspect-oriented programming. Manning, Greenwich, CT, 2003.
21. Liberty, J. Programming C#. O'Reilly, Sebastopol, CA, 2003.
22. Lopes, C., Dourish, P., Lorenz, D. and Lieberherr, K. Beyond AOP: Toward naturalistic programming. ACM SIGPLAN NOTICES, 38 (12). 34-43.
23. Lopes, C.V. and Bajracharya, S., An Analysis of Modularity in Aspect-Oriented Design. Aspect-Oriented Software Development (AOSD'05), 2005 (to appear).
24. Masuhara, H. and Kawauchi, K., Dataflow Pointcut in Aspect-Oriented Programming. Asian Symposium on Programming Languages and Systems (APLAS), 2003, 105--121.
25. Masuhara, H. and Kiczales, G., Modeling crosscutting in aspect-oriented mechanisms. European Conference on Object-Oriented Programming (ECOOP), 2003, Springer, 2-28.
26. Ostermann, K., Mezini, M. and Bockisch, C., Expressive Pointcuts for Increased Modularity. In Proc. of European Conference on Object-Oriented Programming (ECOOP), 2005, Springer.
27. Project, A., AJDT Demonstration, 2004, http://eclipse.org/ajdt/demos/.
28. Rajan, H. and Sullivan, K., Eos: instance-level aspects for integrated system design. Foundations of Software Engineering (FSE), 2003, ACM Press, 297 - 306.

29. Rinard, M., Salcianu, A. and Suhabe, B., A Classification System and Analysis for Aspect-Oriented Programs. Foundations of Software Engineering (FSE), 2004, ACM Press, 147 - 158.
30. Schutter, K.D., What does aspect-oriented programming mean to Cobol? Aspect-Oriented Software Development, 2005, ACM Press, (to appear).
31. Spinczyk, O., Gal, A. and Schröder-Preikschat, W., AspectC++: an aspect-oriented extension to the C++ programming language. Fortieth International Confernece on Tools Pacific: Objects for internet, mobile and embedded applications, 2002, Australian Computer Society, 53 - 60.
32. Sullivan, K.J., Griswold, W.G., Cai, Y. and Hallen, B., The structure and value of modularity in software design. Foundations of Software Engineering, 2001, ACM Press, 99 - 108.
33. Teitelman, W. PILOT: A Step Toward Man-Computer Symbiosis Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1966.
34. Walker, D., Zdancewic, S. and Ligatti, J., A theory of aspects. International Conference on Functional Programming, 2003, ACM Press, 127 - 139.
35. Walker, R. and Viggers., K., Implementing protocols via declarative event patterns. ACM Sigsoft International Symposium on Foundations of Software Engineering (FSE-12), 2004.

Expressive Pointcuts for Increased Modularity

Klaus Ostermann, Mira Mezini, and Christoph Bockisch

Darmstadt University of Technology, D-64283 Darmstadt, Germany
{ostermann, mezini, bockisch}@informatik.tu-darmstadt.de

Abstract. In aspect-oriented programming, pointcuts are used to describe crosscutting structure. Pointcuts that abstract over irrelevant implementation details are clearly desired to better support maintainability and modular reasoning.

We present an analysis which shows that current pointcut languages support localization of crosscutting concerns but are problematic with respect to information hiding. To cope with the problem, we present a pointcut language that exploits information from different models of program semantics, such as the execution trace, the syntax tree, the heap, static type system, etc., and supports abstraction mechanisms analogous to functional abstraction. We show how this raises the abstraction level and modularity of pointcuts and present first steps toward an efficient implementation by means of a static analysis technique.

1 Introduction

In aspect-oriented programming (AOP for short), pointcuts are predicates that identify sets of related points in the execution of a program, where to execute behavior pertaining to crosscutting concerns. Given an aspect that modularizes a crosscutting concern, its pointcuts serve as the interface between the crosscutting concern and the rest of the system. As such, the abstraction level at which these predicates are expressed directly affects the robustness of the design in the presence of change. Separation and localization of concerns into individual units is a major feature of modular design - providing interfaces that absorb local changes is another, equally important, feature.

It has been indicated elsewhere that a pointcut that merely enumerates relevant points in the execution by their syntactic appearance in the program code is fragile w.r.t. changes in the code [15, 20]. In this paper, we investigate the issue in more depth: We compare object-oriented (OO for short) and aspect-oriented (AO for short) designs of an exemplary problem with respect to their capability to remain stable in the presence of change. We observe that with current pointcut languages one can indeed separate crosscutting concerns into their own modular units, but the resulting design does not actually perform much better in terms of absorbing changes than the OO design which does not modularize the crosscutting concerns. This reduces the power of aspects to merely supporting pluggability of crosscutting concerns, leaving out of the reach another important modularity principle: Information hiding [30].

To cope with the problem, this paper proposes a pointcut language that allows to specify pointcuts at a high-level of abstraction by providing (a) different *rich models of the program semantics* and (b) *abstraction mechanisms* analogous to functional abstraction. The key insight is that various models of program semantics are needed to enable reasoning about program execution. For example, the abstract syntax tree (AST) alone is not a very good basis for high-level pointcuts because it is a very indirect representation of the program execution semantics that makes it intractable to specify dynamic properties.

We propose to base the pointcut language on a *combination* of models of the programm's semantics. In this paper, we concentrate on four such models: The AST, the execution trace, the heap, and the static type assignment; if needed, other models such as a profiling or a memory consumption model could be added. Pointcuts in our approach are logic queries over the aforementioned models.

We have implemented a prototype of this approach as an interpreter for a small statically typed AO language, called ALPHA[1]. Pointcuts in ALPHA are logic queries written in Prolog [36]; they operate on-line over databases representing the aforementioned models of the program semantics. We show how AO designs expressed in this language can be made robust against various kinds of changes.

We also present a technique for an efficient implementation of our approach that is based on the notion of *join point shadows* [17]. The shadow of a dynamic join point is a code structure (expression, statement or block) that statically corresponds to an execution of the dynamic join point. The idea is to compute the shadows of pointcuts off-line by a static analysis of pointcuts and to evaluate or extend dynamic semantic models only at these statically computed shadows. Our analysis is different from previous approaches in this direction [17, 28, 35] in that it works on a much more powerful and open pointcut language.

Some concepts used in our approach have also been discussed elsewhere. For example, logic queries have been used in other approaches [15, 19, 37]. The unique contribution of our proposal compared to related work is twofold. First, we present a detailed study of the disadvantages of most current pointcut languages. Second, the openness of the pointcut language, the ability to combine different program models, and the incorporation of the execution trace and heap together with the abstraction mechanisms of a Prolog-like language is also unique. We will give a detailed account of the contribution of this paper and the relation to other works after the technical presentation.

The remainder of the paper is organized as follows. Sec. 2 motivates the need for better pointcut languages by a study of the robustness of aspect-oriented programs. Sec. 3 introduces the ALPHA programming language. Sec. 4 presents some examples in ALPHA and analyzes them in the light of the problems identified in Sec. 2. Sec. 5 describes the static analysis technique. Sec. 6 elaborates on the contribution of this paper in comparison to related work. Sec. 7 describes future work and concludes. The appendix contains different specifications of Prolog constructs that are used in various places but whose specification is not necessary to follow the paper.

2 Pointcuts and Modularity

In this section, we identify the limitations of current pointcut languages by means of an example problem. We focus on AspectJ's pointcut-advice mechanism [21] first; other

[1] Source code is available at [2].

pointcut languages will be discussed in the section on related work. We present an object-oriented (OO) and an aspect-oriented (AO) solution to the problem and compare them w.r.t robustness in the presence of change.

2.1 Example Problem and Its OO and AO Solutions

The example problem is about modeling a hierarchy of graphical objects like points and lines which can be drawn on display objects; each display has a list of figures shown in it. The solution should ensure that the state of figure elements and their corresponding views on active displays is kept synchronized by having displays be updated when the state of figure elements changes.

An OO solution for the problem that applies the observer pattern [14] is schematically shown in Fig. 1[2]. To avoid unnecessary updates, the solution supports what we call *object precision* and *field precision*. By object precision we mean that an update to a figure element triggers a repaint only on those displays on which the figure element is visible; in general, there are multiple different display objects active, whereby every figure element is visible only on a (possibly empty) subset of all displays. For this purpose, each figure in Fig. 1 maintains an observer list with the displays it is shown in, if any; when a figure `f` is added to a display, the display is added to the list of `f`'s observers as well as to the observer lists of `f`'s children; e.g., showing a line on a display will cause the display to be an observer of the line as well as of its start and end points. If a figure `f1` is not anymore a child of another figure, `f2`, the observers that `f1` inherited from `f2` are removed from the list of `f1`'s observers[3].

By field precision we mean that only changes of the fields that contribute to the graphical representation of a figure element should trigger display updates. The set of the fields affecting the draw behavior generally depends on the dynamic control flow and cannot be determined statically. Hence, it is not always easy to ensure field precision especially if the system is complex. In the `Line` class in Fig. 1, it is easy to see that the field `name` is never involved in the drawing behavior and that `enable`, `start` and `end` are potentially read in the control flow of `draw`. Of the latter variables, only `enabled` is always read - hence, a change to it always triggers a notification of the observers; fields `start` and `end` are only read if `enabled` is true. Hence, changes to `start` or `end` trigger a notification only if `enabled` is true (see comments on the methods `notifyObserversUnconditional()`, `notifyObservers()` and `setEnabled(...)` in class `FigureElement`, and `Line.setEnd(...)` in Fig. 1).

A functionally equivalent AO solution of the problem is schematically shown in Fig. 2. This solution factors the observer management fields and methods out of the figure element classes into the aspect using the inter-type declaration mechanism of AspectJ[4]. The aspect defines three pointcuts. The pointcut `addFigure` captures any call

[2] Complete code for all examples and our ALPHA interpreter are available at [2].

[3] If figures are shared by several parent-figures, reference counters are associated with observers and an observer is actually removed from an observer list, only if its reference counter is zero.

[4] The observer implementation proposed by Hannemann and Kiczales [16] uses hashtables instead of introductions in order to increase the reusability of aspects, but this does not affect the discussion in this paper.

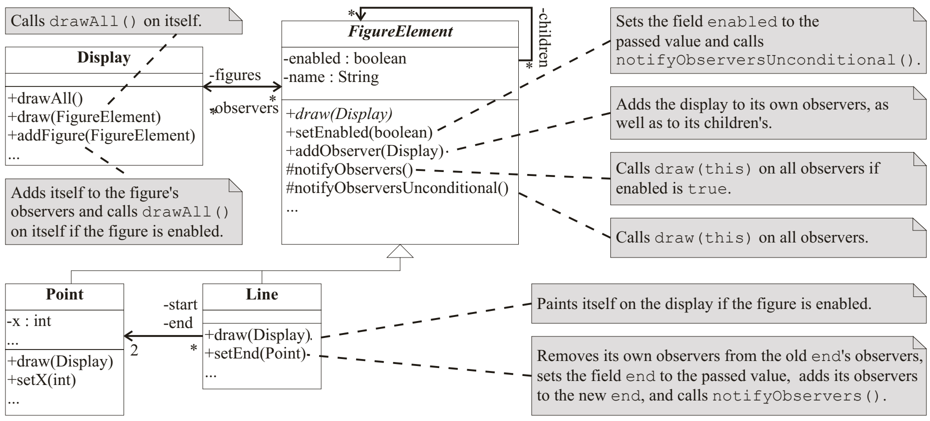

Calls drawAll() on itself.
Display
+drawAll()
+draw(FigureElement)
+addFigure(FigureElement)
...
-figures
*observers
*
Adds itself to the figure's observers and calls drawAll() on itself if the figure is enabled.
FigureElement
-enabled : boolean
-name : String
+draw(Display)
+setEnabled(boolean)
+addObserver(Display)
#notifyObservers()
#notifyObserversUnconditional()
...
-children
*
*
Sets the field enabled to the passed value and calls notifyObserversUnconditional().
Adds the display to its own observers, as well as to its children's.
Calls draw(this) on all observers if enabled is true.
Calls draw(this) on all observers.
Point
-x : int
...
+draw(Display)
+setX(int)
...
-start
-end
2
*
Line
+draw(Display)
+setEnd(Point)
...
Paints itself on the display if the figure is enabled.
Removes its own observers from the old end's observers, sets the field end to the passed value, adds its observers to the new end, and calls notifyObservers().

Fig. 1. OO implementation of a precise version of the display updating

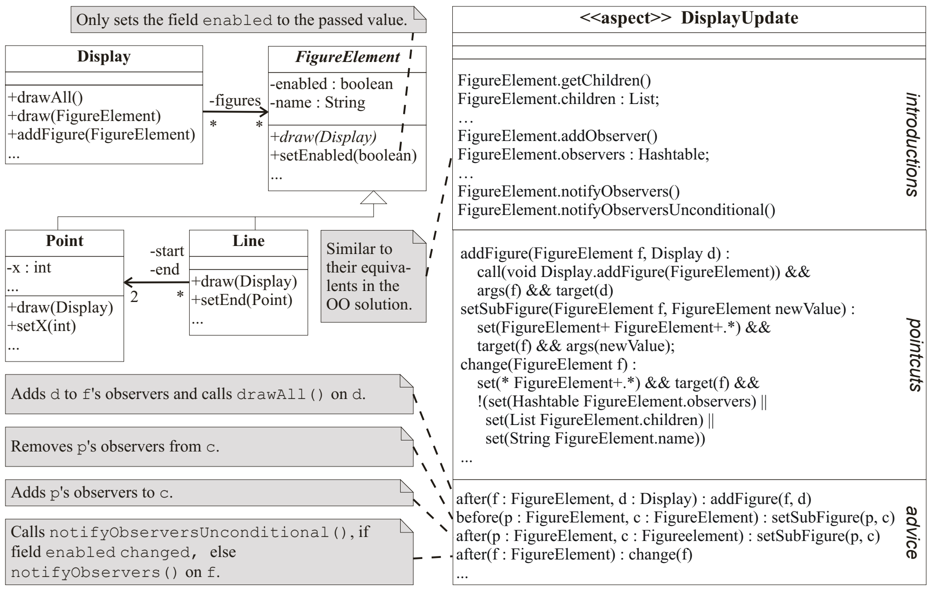

Only sets the field enabled to the passed value.
Display
+drawAll()
+draw(FigureElement)
+addFigure(FigureElement)
...
-figures
*
*
FigureElement
-enabled : boolean
-name : String
+draw(Display)
+setEnabled(boolean)
...
Point
-x : int
...
+draw(Display)
+setX(int)
...
-start
-end
2
*
Line
+draw(Display)
+setEnd(Point)
...
Similar to their equivalents in the OO solution.
Adds d to f's observers and calls drawAll() on d.
Removes p's observers from c.
Adds p's observers to c.
Calls notifyObserversUnconditional(), if field enabled changed, else notifyObservers() on f.
<<aspect>> DisplayUpdate
FigureElement.getChildren()
FigureElement.children : List;
...
FigureElement.addObserver()
FigureElement.observers : Hashtable;
...
FigureElement.notifyObservers()
FigureElement.notifyObserversUnconditional()
introductions
addFigure(FigureElement f, Display d) :
call(void Display.addFigure(FigureElement)) &&
args(f) && target(d)
setSubFigure(FigureElement f, FigureElement newValue) :
set(FigureElement+ FigureElement+.*) &&
target(f) && args(newValue);
change(FigureElement f) :
set(* FigureElement+.*) && target(f) &&
!(set(Hashtable FigureElement.observers) ||
set(List FigureElement.children) ||
set(String FigureElement.name))
...
pointcuts
after(f : FigureElement, d : Display) : addFigure(f, d)
before(p : FigureElement, c : FigureElement) : setSubFigure(p, c)
after(p : FigureElement, c : Figureelement) : setSubFigure(p, c)
after(f : FigureElement) : change(f)
...
advice

Fig. 2. First AO implementation of the precise display updating

to the method `Display.addFigure(FigureElement)`; the after-advice associated with this pointcut establishes a subject-observer relation between the receiver and the argument.

The `setSubFigure` pointcut captures points in the execution, where parent-child associations are changed - these are assignments on any field of type `FigureElement` or a subtype thereof (denoted by the "+"), declared in `FigureElement` or any of its subclasses. The before and after advice associated with this pointcut make sure that the observer lists are updated accordingly. The `change()` pointcut captures assignments to those fields of figure element objects that affect the draw behavior - the set of relevant fields includes any field declared in `FigureElement` or one of its subclasses, excluding the fields `FigureElement.name`, `FigureElement.observers` and `figureElement.children` (the latter two are introduced by the aspect). The advice associated with this pointcut ensures that notifications are sent to relevant observers.

2.2 Comparison of the OO and AO Solutions

The main advantage of the AspectJ solution over the OO counterpart is that the display updating protocol is made explicit and *localized* in one module. Due to this separation changes to the display update protocol are localized within the aspect. For example, assume that we decide to modify the protocol as follows. The display update signaling currently performed within methods that change the state of figure elements, should happen at caller sites of these methods (e.g., the because the caller object should be logged, which is not possible at execution site). Changes needed to introduce the modified protocol are localized within the aspect code in the AO solution (alternatively a new aspect can be implemented); the same changes are not localized in the OO solution. Furthermore, the separation makes the display updating logic pluggable. The advantages resulting from the separation of crosscutting concerns are discussed elsewhere [22, 16] and are not in the focus of this paper.

Physical separation and localization of concerns, while important, is only one aspect of modularity. Another, equally important aspect of modularity is about the interface that controls the interaction of the separated logic with the rest of the system, thereby employing abstraction mechanisms to hide implementation details. The interface of the separated observer protocol to the rest of the system is defined by pointcuts in the aspect in Fig. 2. A recent paper by Kiczales and Mezini [23] argues that this explicit interface makes modular reasoning in the presence of change easier in an AO setting compared to an OO setting, where there is no explicit interface between these two concerns.

In this paper, we go one step further and investigate the ability of the AO interfaces to absorb change by means of information hiding. Unfortunately, interfaces supported by current pointcut technology fall short in this respect. The set of points in the execution of figure elements where to update appropriate display objects are not defined intentionally by some common semantic property, say, as points where *”changes occur on fields that were previously read in the control flow of the last* `drawAll` *call”*. Rather, the pointcuts in our example mostly describe these points by their syntax, thus, exposing implementation details of the figure element hierarchy to the aspect. The following comparison of the OO and AO solutions from Fig. 1 and Fig. 2 shows that the lack of proper support for information hiding makes the separated display update protocol basically as fragile w.r.t changes as the OO solution. The comparison is organized around the change scenarios presented in Fig. 3, which also summarizes the robustness of the AO and OO solution related to these change scenarios.

First, both scenarios are fragile with respect to scenarios *Ch1* and *Ch3*. In both solutions, moving parts of a figure element’s state to a helper class will cause changes to those fields to escape observation, although they might have had effect on the drawing behavior. Hence, they break w.r.t. *Ch1*. Also, renaming the field `enabled`, or adding a

name	description	example	OO	AO
Ch1	Object graph change: Outsource part of the drawing relevant state of a figure element to a class that is not in the `FigureElement` hierarchy.	Use an object of type `Pair` to store the coordinates of a `Point`.	–	–
Ch2	Class hierarchy change: Inserting a new type into the hierarchy of `FigureElements`.	Adding the class `Circle extends FigureElement`.	+/–	+/–
Ch3	Control flow change: Change the condition under which a display update is necessary.	Renaming the field `enabled` to `visible`, or adding a field `hidden`.	–	–
Ch4	Class definition change: Inserting/removing a field whose change makes display update necessary.	Adding the field `color` to the class `FigureElement`.	–	+
Ch5	Class definition change: Inserting/removing a field whose change does not affect display.	Adding the field `changeHistory` to the class `FigureElement`.	+	–

Fig. 3. Change scenarios with comparison of AO and OO solution

new field which also controls when figure elements are displayed, say `hidden`, will break both the AO and OO protocols. This is because the names of such fields are hard-coded in the implementation of `notifyObservers()`, which is the same in both solutions. Hence, both solution fail to absorb *Ch3*.

The AO solution is more robust w.r.t *Ch4*. The OO protocol breaks in the sense that the display update signaling for the field being added, needs to be adopted, respectively encoded anew. The AO protocol that uses wildcards for pattern matching on names of fields that affect drawing behavior carries over automatically. However, the AO solution is less robust than the OO solution w.r.t *Ch5*. This is because the `change` pointcut in Fig. 2 enumerates each field to exclude from the observation explicitly. Adding (or removing) a field which does not influence the graphical representation of a figure will break field precision of the aspect: Changes to these fields will cause the display to be updated.

Finally, with regard to the scenario *Ch2*, we argue that both solutions perform more or less the same under the assumption that the new class, in general, introduces both fields that affect the drawing behavior as well as fields that do not affect drawing. The AO solution performs better for fields that affect drawing: The protocol established by the aspect automatically applies to the new class. This is not true for the OO solution: The whole logic concerning children and field change should be manually coded in the new class. However, the opposite is true for fields that do not affect drawing and, hence, need to be excluded explicitly in the AO solution.

The use of wildcards for pattern matching on names might at first sight appear to support some sort of abstraction by providing a means to identify relevant execution points by some commonality. However, pattern matching on names only allows to abstract over syntax, which is not always sufficient. Our investigation shows that wildcards do not actually increase the ability to absorb change, beyond simple cases, where there are no exceptions to be made from the rule defined by wildcards. As for our example, we could as well have used an AO solution that does not use wildcards but enumerates the relevant points. This solution would exhibit the same robustness w.r.t the change scenarios as the OO solution[5].

The discussion suggests that without more powerful mechanisms for information hiding the potential of AO mechanisms for improving modularity cannot fully be unleashed. This has been the motivation for us to work on a pointcut language that enables better modularity and information hiding. This language will be presented in the following section. Please note that the question is not whether AO mechanisms provide better modularity than OO mechanisms; the question is rather how to further improve the power of the modularity of AO mechanisms. As mentioned in the beginning of this sub-section, AO does support better modularity by separating and localizing the display update logic [22, 16] and by providing an explicit interface [23]. The point we want to improve is that the focus of pointcuts should be *when* (under which conditions) a pointcut should be triggered rather than *where* (lexically) the corresponding places in the code are.

3 The ALPHA Language

ALPHA is an AO extension of a toy OO core language implemented as an interpreter in Java. The OO core of ALPHA is based on L2 [12] - a simple object-oriented language in the style of Java. The formal syntax, semantics, and type system of L2 are described in [12]. Here we present the OO core of ALPHA informally by means of the example in Fig. 4 - a simplified variant of the example from the previous section. ALPHA supports classes and single inheritance and has a standard static type system.

3.1 Pointcuts and Advice

Every class in ALPHA may define fields and methods and may also define pointcuts and associate advice with them. Pointcuts are Prolog queries over a database of both static and dynamic information about the program or program execution. A Prolog query is a sequence of primitive queries combined by the *and* operator "`,`". A simple pointcut that denotes "all assignments to fields of objects of type point" is shown in the class `DisplayUpdate` in Fig. 5.

In contrast to AspectJ and similar to Caesar [29], aspects in ALPHA become effective only after they are *deployed*. Further discussion of this strategy is available at [29]. What matters is to note that the advice of `DisplayUpdate` will have semantic effect once an instance of `DisplayUpdate` is deployed. For illustration, consider the use of the aspect `DisplayUpdate` within the `main()` method of class `Main` in Fig. 5 - here, the advice of `DisplayUpdate` will be effective only during the execution of `doSomething()`.

In order to explain the pointcut in Fig. 5, it is necessary to understand the basic structure of the database. The database contains both static and dynamic information about the program organized in a set of relations. A very simple relation is the unary relation `now`, denoted `now/1`. This relation has only one fact that contains the current

[5] In [2], the reader can also find code for an AspectJ solution of the example problem that does not uses wildcards.

```
class FigureElement extends Object {
  String name;
  void draw(Display d) {}
}
class Point extends FigureElement {
  int x, y;
  boolean enabled;
  void draw(Display d) {
    if( this .enabled) d. paintPoint ( this .x, this .y);
  }
}
class Line extds FigureElement {
  Point start , end;
  void draw(Display d) {
    if( this .enabled) d. paintLine ( this . start , this .end);
  }
}
class Display extends object {
  FigureElement f1 , f2;
  void drawAll() { this . f1 .draw( this ); this . f2 .draw( this ); }
  void draw(FigureElement fe ) { print (" display update :"); print (fe ); this .drawAll (); }
  void paintPoint ( int x, int y) { ... }
  void paintLine (Point start , Point end) { ... }
}
```

Fig. 4. Figure elements in ALPHA

```
class DisplayUpdate extends Object {
  Display d;
  after now(ID), set (ID, _, P, _, _), instanceof (P, 'Point') { this .d.draw(P); }
}

class Main extends Object {
  Display d; DisplayUpdate du;
  void main() {
    this .d = new Display (); this .du = new DisplayUpdate();
    this .du.d = this .d;
    deploy( this .du) { this .doSomething() }
  }
  void doSomething() { ... }
  ...
}
```

Fig. 5. Simple advice in ALPHA

timestamp. These timestamps are necessary in order to reason about temporal relations between events. The query `now(ID)` in Fig. 5 retrieves the current timestamp and binds it to the variable `ID`. In Prolog, all names starting with uppercase letters are considered variables, whereas all names starting with lowercase letters or enclosed by single quotes (') are considered constants.

The second part of the query, `set(ID, _, P, _, _)`, queries a relation `set/5` that stores all assignments in the current execution. The first element of this relation is the timestamp of this event, the second one is a reference into the syntax tree of the program and denotes the expression in the syntax tree that corresponds to this event. The third element is the object that contains the field, the fourth is the fieldname, and the fifth is the value assigned to the field. By using the name `ID` for the first element in the query `set(ID, _, P, _, _)`, we specify that this pointcut will match only

```
class DisplayUpdate extends Object {
  Display d; Point p;
  after now(ID), set(ID, _, @this.p, _, _) {
    this.d.draw(this.p);
  }
}
```

Fig. 6. Inserting context into a pointcut

if the assignment has happened right now and not in some time in the past because a variable (`ID` in this case) must be bound to the same value in all places where it is used. The wildcard "_" is used for anonymous variables that are not interesting; in the set part of the pointcut in Fig. 5 the "_" wildcards are used to denote that the pointcut matches for any expression as well as for any field name and value being assigned. By using the name `P` for the receiver element of the `set/5` relation, we bind the receiver object of each matching assignment to `P`. In the third part of the pointcut, `instanceof(P, 'Point')`, we constrain the set of eligible assignments even further by requiring `P` to be an instance of the class `Point`.

Variables in a pointcut can be used both as constraints during the unification process and as a means to make context available in the advice. In our example, we use `P` in the advice body[6]. The form of advice is similar to AspectJ [21].

The kind of data in the database is obviously an important variation point of our approach. In our prototype, the database contains four different program models: a representation of the abstract syntax tree, a representation of the object store (heap), a representation of the static type of every expression in the program, and a representation of the trace of the program execution. These four structures are a natural choice, since they represent the main entities used for interpreting a program. However, it would not invalidate our approach to add or remove other entities, e.g., add a model about resource consumption or remove the object store model. In the example above, the `set/5` relation belongs to the execution trace model, whereas the `instanceof/2` relation belongs to the object store model and the static type model. A full reference of the relations in the database is available in the appendix in Fig. 14. The basic idea is that each of these models represents a partial view of the program semantics. By making as much information available to the pointcut programmer as possible, the programmer can choose the program model to be used to express his intention as directly and conveniently as possible.

The escape symbol `@` can be used to evaluate an expression inside a query, such that object-specific constraints involving values from the enclosing object can be expressed. Fig. 6 shows a refined version of the display update aspect whose pointcut will match only if the target of the assignment is `this.p`.

3.2 Pointcut Abstraction and Pointcut Libraries

The expressiveness of our pointcut language is due to the rich program models *and* its abstraction mechanisms. Due to Prolog, it is easy to define new predicates that abstract

[6] We use type inference to determine a static type for every variable inside a query, which is used to type-check the advice body. We elaborate on this in Sec. 5.

over the primitive generated predicates. Fig. 7 shows an excerpt of the standard pointcut library building on this feature. Line 3 shows - by the example of the primitive `set` predicate - how to define convenient abbreviations of the generated primitive execution trace predicates for the case that we are not interested in past events or in the syntactic location of the event. With these abbreviations the pointcut in Fig. 6 can be written more conveniently as `set(@this.p,_,_)`.

Lines 6-9 demonstrate the usefulness of having the complete history of execution[7]. The `cflow` query specifies under which conditions a join point `ID0` is or has been in the control flow of another join point `ID1`. Reasonably, this other join point may only be a method call. Thus, `ID1` must be the timestamp of a method call. Method calls are stored in the database as pairs of `calls/5` and `endcall/3` facts denoting the beginning respectively the end of a method call. The `before/2` relation is a part of the execution trace model and can be used to compare events w.r.t. their temporal order. The first rule of the `cflow` query, applies when the `ID1` call has completed (i.e., a corresponding `endcall` fact is available). The second rule applies when `ID1` is still on the call stack; it uses the special control predicate `\+`, which succeeds, if the goal cannot be proven (a.k.a. *negation as failure*).

Please note that this `cflow` construct is much more powerful than the AspectJ pointcut designator with this name, since the AspectJ variant can only be used to refer to control flows that are currently on the call stack (corresponding to our second `cflow` rule), whereas our `cflow` also applies to control flows in the past[8].

The pointcut library contains a set of other pointcut predicates that are only sketched in Fig. 7. We have defined predicates to determine whether an object is reachable from another object following a path of links in the object graph (`reachable/2`) and to determine the class of an object (`instanceof/2`). Other predicates provide convenient access to the AST (`class/2`, `method/3`, `field/3`, `within/3`, `subtype-eq/2`).

The `pcflow/3` predicate predicts the control flow of a method based on the AST, basically building the call graph of the method. This is achieved by computing the transitive hull of all outgoing method calls, whereby all method implementations in all subtypes are considered in order to take late binding into account.

The `mostRecent/2` predicate finds the most recent occurence of an event pattern. We will use this predicate to express things like "find the most recent occurence of a call to `draw`".

A very convenient property of Prolog is that these predicates can be used with arbitrary instantiation patterns. This means that the predicates can be used in any direction. For example, the `within/3` predicate can be used to find an expression within a given method and class, or the other way around, to find a class and a method that lexically contain an expression.

[7] Due to the optimizations discussed in sec. 5 only parts of the execution history are recorded that are relevant for the pointcuts in the program.

[8] Note, however, that the main purpose of this paper is *not* to propose new control flow pointcut designators. The `cflow` pointcut designator is just an illustration of the extensibility of our pointcut language.

```
% abbrevations if we are only interested in the current event
set(Receiver, Field, Val) :− now(ID), set(ID, _, Receiver, Field,
Val).
% abbreviations for new, calls, get similarly

% is ID0 in the control flow of ID1?
cflow(ID0, ID1) :− calls(ID1, _, _, _, _), before(ID1, ID0),

                    endcall(ID2, _, ID1, _), before(ID0, ID2).
cflow(ID0, ID1) :−
   calls(ID1, _, _, _, _),  before(ID1, ID0), \+ encall(_, _, ID1, _).

% is Obj2 reachable from Obj1 in the object graph?
reachable(Obj1,Obj2) :− ...

% is Obj an instance of C?
instanceof(Obj, C) :− ...

% convenient access to AST: query classes, methods, and fields
class(Name, CDef) :− ...  meth(CName, MName, MDef) :− ...
field(CName, FName, FDef) :− ...

% is Expr within method MName of class CName
within(Expr, CName, MName,) :− ...

% is C1 subtype of C2?
subtypeeq(C1, C2) :− ...


% is Expr in the statically predicted control flow of CName.MName?
pcflow(CName, MName, Expr) :− ...

% find the most recent event matching a pattern X
mostRecent(ID,X) :− ...
```

Fig. 7. Excerpts of the pointcut library

The full definition of these predicates can be found in Fig. 15 in the appendix. For the purpose of this work, the details of their definition are not very important. The interesting point is that ALPHA has an *open* pointcut language, whereby new pointcuts can be added on-demand in a *declarative* way. Simple pointcuts can be combined to more powerful pointcuts, so we have the same kind of abstraction mechanism for pointcuts that functional abstraction provides for functions.

We have developed a rudimentary module mechanism for pointcut libraries. Currently, we have a standard pointcut library, that is always available, and user-defined pointcut libraries, that must have the same file name as the source file. A pointcut library can import other libraries using Prolog's own module mechanism. It would be a straightforward extension to make this a full-fledged module mechanism with explicit imports and exports, namespaces, etc.

4 Programming with ALPHA

In this section, we demonstrate how information from different program models can be combined to increase the abstraction level of pointcuts. Furthermore, we discuss the implications of our pointcut language on the programming model.

4.1 Expressiveness of Pointcuts

The class `DisplayUpdate` in Fig. 8 shows six different ways to specify a display update pointcut in ALPHA, using different models of the program. We use these six differents pointcuts in order to show how we can gradually increase the abstraction level of the pointcuts by exploiting the available information in the database. The resulting pointcuts differ in their support for *robustness* and *precision*, as discussed below and summarized in Fig. 9.

```
 1 class DisplayUpdate extends Object {
 2   Display d;
 3
 4   // enum pointcut
 5   after set(P, x, _); set(P, y, _); set(P, 'start', _); set(P, 'end', _),
 6         instanceof(P, 'FigureElement') { this.d.draw(P); }
 7
 8   // set* pointcut
 9   after set(P, _, _), instanceof(P, 'FigureElement') { this.d.draw(P); }
10
11   // pcflow pointcut
12   after now(ID), set(ID, ExpID1, P, F, _), instanceof(P, 'FigureElement'),
13         pcflow(Display, 'drawAll', (_, get((ExpID2, _), F))),
14         hastype(ExpID2, 'FigureElement') { this.d.draw(P); }
15
16   // cflow pointcut
17   after set(P, F, _), get(T1, _, P, F, _), mostRecent(T2, calls(T2, _, @this.d,'drawAll', _)),
18         cflow(T1, T2), instanceof(P, 'FigureElement') { this.d.draw(P); }
19
20   // cflowreach pointcut
21   after set(P, F, _), get(T1, _, P, F, _), mostRecent(T2, calls(T2, _, @this.d,'drawAll', _)),
22         cflow(T1, T2), reachable(Q, P), instanceof(Q, 'FigureElement') { this.d.draw(P); }
23 }
```

Fig. 8. Six display update pointcuts

The `enum` pointcut (line 5, Fig. 8) enumerates[9] all assignments to fields that potentially effect drawing behavior, namely to fields `x`, `y`, `start`, or `end` of any object `P` of type `FigureElement`. It uses the names of the fields to identify the relevant assignments. By precisely enumerating the fields potentially involved with drawing, the pointcut supports some sort of static field precision: It makes at least sure that changes to fields that are never read in any control flow of `drawAll()` do not trigger display updates. However, it requires the programmer to explicitly encode this knowledge. Furthermore, it does not take into account the actual control flow of the concrete program execution and, hence, cannot avoid e.g., updates after assignments to fields of disabled points. Also, object precision is not supported. Precision w.r.t. fields involved in the drawing behavior only under certain dynamic conditions - for convenience let us call this dynamic field precision - and object precision require knowledge from dynamic program models, which this pointcut does not make use of. With respect to robustness, the `enum` pointcut exhibits the same behavior as the OO solution in Sec. 2.

The `set*` pointcut (line 9) is triggered by assignments to *any* field of a `FigureElement` object. Due to the use of the _ wildcard this pointcut may cover too many execu-

[9] A semicolon denotes "or" in Prolog

criteria	enum	set*	pcflow	cflow	cflowreach
static field precision	+	-	+	+	+
dynamic field precision	-	-	-	+	+
object precision	-	-	-	+	+
Ch1	-	-	-	-	+
Ch2	+/-	+/-	+	+	+
Ch3	-	-	+	+	+
Ch4	-	+	+	+	+
Ch5	+	-	+	+	+

Fig. 9. Evaluation of pointcuts w.r.t. change scenarios from Fig. 3

tion points whose signature matches the pattern by accident [15, 20, 13]. As a result, the pointcut performs poorly w.r.t. field precision: Any assignment to the field `name` which is not at all involved with drawing will also trigger a display update. Similar to `enum`, `set*` uses only static information, hence, it supports neither dynamic field precision nor object precision. As far as robustness is concerned, `set*` exhibits the same behavior as the AO solution in Sec. 2.

The `pcflow` pointcut (line 12) uses the `pcflow` predicate to approximate the control flow of `Display.drawAll()` based on the AST model and selects field read expressions in the approximated control flow; only assignments to such fields match the `pcflow` pointcut. Similar to `enum`, this pointcut ensures that changes to fields that are never read in the control flow of `drawAll()` do not trigger display updates. However, neither object nor dynamic field precision is supported, since the pointcut only makes use of the AST and not of the dynamic models of the program. The pointcut is not robust in the case of scenario *Ch1* - outsourcing part of figure element state to external objects. The pointcut explicitly requires `P` to be a `FigureElement` in order to be able to pass it to the `draw()` method call. So, state outsourced to non `FigureElement` objects escape the observation by this pointcut.

Note that while supporting the same precision as `enum`, `pcflow` is much more robust. This is due to the abstraction capabilities of the pointcut language (including functional composition and higher-order pointcuts), which allows us to compose primitive pointcuts into more powerful ones, such as `pcflow`. With a pointcut language that does not support such abstraction mechanisms, e.g., AspectJ's pointcut language that only provides operations on sets - union (||), intersection (`&&`), negation (`!`) -, the programmer cannot express the intention to "first identify all field accesses in the control flow of a certain method and than select set operations to these fields" in terms of a generic description, if this functionality is not available as a primitive pointcut designator. (S)he is basically left with the explicit enumeration of such field accesses, as in `enum`; the only alternative to enumeration is to describe general rules by wildcards, which is actually not better with regard to robustness.

The `cflow` pointcut (line 17) is similar to `pcflow` in that this pointcut also selects field reads in the control flow of `drawAll`. The crucial difference is that `cflow` is based on the actual control flow at runtime rather than on a conservative static approximation of it. As a result, `cflow` performs better than `pcflow`. It supports both dynamic and static field precision as well as object precision: Only assignments to a field `F` of an object `P` that are read in the control flow of the particular display object denoted by `this.d` (see the expression `@this.d` in the `pcflow` pointcut) trigger an update. Changes of any field of any figure element that is not referred to by our active display denoted by `this.d` do not trigger updates. By its use of the dynamic execution model of the program, `cflow` significantly improves over `pcflow`. Note that it would not be possible to express something similar with the AspectJ `cflow` construct because the `drawAll` method call is in the past and not on the call stack. The only problem with `cflow` is lack of robustness w.r.t. *Ch1*.

The `cflowreach` pointcut (line 21) solves the robustness problem of `cflow` w.r.t *Ch1*. This pointcut composes the `cflow` pointcut with the `reachable` predicate from Fig. 7/Fig. 15. That is, in addition to assignments to objects of type `FigureElement`, it also captures assignments to any object that is reachable in the object graph from an instance of `FigureElement`. The use of the object graph model makes `cflowreach` robust against *Ch1*. Since it also inherits all features of the `cflow` pointcut, `cflowreach` fulfills the precision requirements and is robust w.r.t. all change scenarios *Ch1* to *Ch5*.

The foregoing analysis demonstrates how our approach enables robust and precise pointcuts. The pointcuts `cflow` and `cflowreach` above encode minimal knowledge about implementation details of the crosscutting structure they describe. They directly express the semantic properties of the display update structure rather than relying on implementation details of how the latter syntactically appears in the program code (the names of the variables involved with drawing are irrelevant for the display update behavior).

This is due to the rich models of program semantics underlying these pointcuts as well as the abstraction mechanisms of the pointcut language. In our approach, the programmer can, however, choose which models of the program (s)he wants to use to express a pointcut: from pure syntactic to very dynamic, operational properties, whichever describe the crosscutting best. In this context, please also note the role of unification in elegantly expressing relations between join points. This is illustrated e.g., by the `cflow` pointcut, where unification together with the `cflow` predicate is crucial in expressing the temporal relation between points where variables are read respectively written in the execution flow of `drawAll`.

4.2 Expressive Power, Openness, and Simplicity

In this section, we reason about the complexity of the programming model of our pointcut language. We argue that in addition to increasing the expressiveness of the language, the rich models of program semantics and the powerful abstraction mechanisms such as Prolog's unification also decrease the complexity of the programming model.

First, consider the version of ALPHA, call it `Fixed-ALPHA` with a fixed pointcut language, including e.g., only the predicates defined in our standard library. The expressiveness of this language is increased as compared to AspectJ - all pointcuts in Fig.8 are written in this language. Nonetheless, the programming model is not more complex than that of AspectJ-like languages [21]. Similar to AspectJ, the programmer needs to understand the meaning of some predefined pointcuts, such as `cflow`, `within`, etc., as well as the semantics of Prolog operators/unification for composing them.

Now let us consider the full ALPHA language, in which (domain-specific) pointcut libraries can be defined as outlined in Sec. 3.2. One may argue that this introduces the complexities of full meta-programming into AOP. Similar to [15], we argue that the problems with full meta-programming occur only in an imperative type of language where the programmer is directly involved with some sort of program transformation.

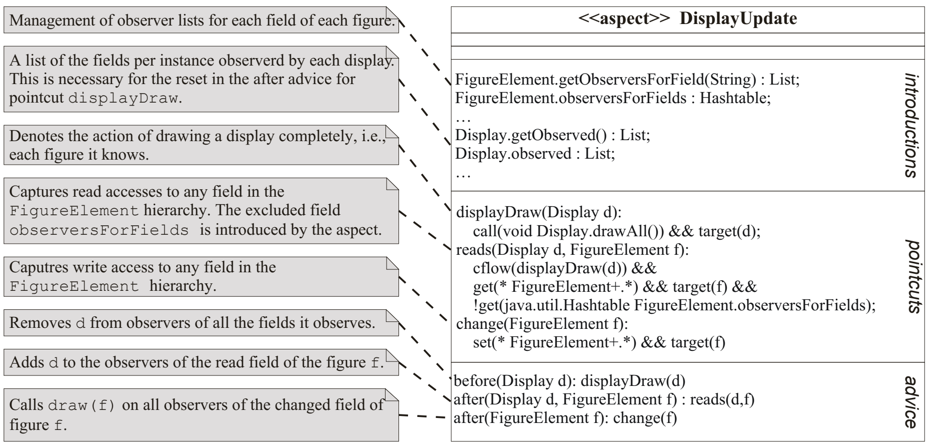
With ALPHA, the programmer only specifies where and what behavioral effect to apply and is not concerned with how this effect is achieved in terms of operational details. To support our argumentation, two examples are discussed in the following which demonstrate that richer program models and more powerful abstraction mechanisms decrease rather than increase the complexity of the programming model.

First, we review the pointcuts `cflow` and `pcflow` from Fig. 8. They both identify assignments involved in the display update crosscutting by their property of accessing variables previously read in the control flow of `drawAll`. However, the models they use are different. The `cflow` predicate uses a richer model that includes the execution trace; `pcflow`'s model is the AST on top of which it approximates the dynamics. We already argued in Sec. 4.1 that `cflow` specifies the crosscutting structure more precisely. Nonetheless, `cflow` is less complex and easier to understand than `pcflow` (see respective definitions in Fig. 15); The approximation of the dynamics of execution based on the AST model adds accidental complexity to `pcflow`'s definition.

Second, we compare the ALPHA implementation of display updating using the `cflow` pointcut in Fig. 8, with the AspectJ solution shown in Fig. 10. The latter is operationally equivalent to the former: it tries to express the rule *"whenever changes are performed on fields that were previously read in the control flow of the last* `drawAll` *call, make an update"* by quantifying over the dynamic control flow. However, to compensate for the lack of the needed information about the dynamic execution trace, a model of the latter is constructed and managed by the programmer within the aspect. Especially, in lack of more powerful abstraction mechanisms beyond operations on sets, building this model employs the imperative Turing-completeness of Java.

Concretely, the aspect administers observer lists for individual fields rather than for whole objects; an instance field of type `Hashtable` is added into the class `FigureElement`, whose keys are field names and whose values are the corresponding lists of observers, i.e., `Display` objects. A display is made an observer of those fields that have been read during the last execution of its `drawAll()` method (see the after advice associated with the pointcut `reads` in Fig. 10). The pointcut `change` captures assignments to fields of figure elements binding the receiver object to `f`; the after advice associated with it uses `f` together with the name of the assigned field to retrieve displays that observe the field, if any. Before calling `draw` on each observer display, the latter is removed from all observer lists it is in, since different fields might get read during the next draw (field precision).

Like the `cflow`-based solution in ALPHA, the implementation in Fig. 10 is robust w.r.t. all change scenarios (but *Ch1*). However, the aspect schematically shown in Fig. 10 is very complex as compared to the pointcut-advice `cflow` in Fig. 8. Instead of declaratively defining the crosscutting structure employing functional abstraction, as its ALPHA counterpart does, the aspect employs the imperative Turing-completeness


Management of observer lists for each field of each figure.
A list of the fields per instance observerd by each display. This is necessary for the reset in the after advice for pointcut displayDraw.
Denotes the action of drawing a display completely, i.e., each figure it knows.
Captures read accesses to any field in the FigureElement hierarchy. The excluded field observersForFields is introduced by the aspect.
Caputres write access to any field in the FigureElement hierarchy.
Removes d from observers of all the fields it observes.
Adds d to the observers of the read field of the figure f.
Calls draw(f) on all observers of the changed field of figure f.
<<aspect>> DisplayUpdate
introductions
pointcuts
advice

```
FigureElement.getObserversForField(String) : List;
FigureElement.observersForFields : Hashtable;
…
Display.getObserved() : List;
Display.observed : List;
…

displayDraw(Display d):
   call(void Display.drawAll()) && target(d);
reads(Display d, FigureElement f):
   cflow(displayDraw(d)) &&
   get(* FigureElement+.*) && target(f) &&
   !get(java.util.Hashtable FigureElement.observersForFields);
change(FigureElement f):
   set(* FigureElement+.*) && target(f)

before(Display d): displayDraw(d)
after(Display d, FigureElement f) : reads(d,f)
after(FigureElement f): change(f)
```

Fig. 10. More robust AO implementation of the precise display updating

of Java to build up a complex infrastructure to basically reverse-engineer the dynamic execution; trying to make it robust w.r.t *Ch1* will further increase the complexity.

All the above said on the decreased rather than increased complexity of the programming model, we would like to add that further investigation is still in place to judge whether the Turing completeness of Prolog is actually needed. It would clearly be desirable to have a simpler, but still sufficiently expressive, pointcut language, both for further decreasing the complexity of the programming model as well as for making an efficient implementation of the language simpler. We did not, however, want to restrict the expressiveness of our language from the very beginning and will consider this issue in future investigation.

5 Abstract Interpretation of Pointcuts

A naive implementation of our approach that extends the Prolog database and evaluates all pointcuts after *every* computation step is obviously not acceptable from both *time* and *space* perspectives.

In this section, we present a new static analysis technique that evaluates pointcuts statically in order to compute (a) a (small) set of expressions in the AST (i.e., join point shadows) that will potentially influence the result of a pointcut, and (b) the *lifetime* of facts that are generated at these shadows. The interpreter can take advantage of this information by extending the database and evaluating the pointcuts only if an expression from the aforementioned pre-computed set is evaluated, and by discarding data in the database if its lifetime is over. A side-effect of the static analysis is that it also infers static types for query variables used to type check advice bodies.

Our optimization is based on an abstract interpretation [8] of the pointcuts. Abstract interpretation of a program uses its denotation to make computations in a universe of abstract objects so that the result of an abstract execution gives some information on the actual computation [8].

Domain	**Static abstraction**
Time stamps	{now,past}× Expression IDs
Values	Types
Execution Trace	Virtual Trace
Object Store	Virtual Store

Fig. 11. Runtime domains and their static abstractions

In our case, we approximate the runtime domains shown in Fig. 11. The interpretation is done by a special Prolog interpreter (written in Prolog itself) that evaluates pointcut queries based on our abstract domains and collects data about join point shadows and lifetime during the interpretation.

The virtual trace defines all predicates from the execution trace as rules over the abstract syntax tree and the static type model. For illustration we will only consider the `calls/5` predicate. In a similar way, all other predicates of the execution trace are approximated statically. Their exact definition can be found in the appendix in Fig. 16 and on the project website [2]. The `call/5` predicate is defined as follows:

```
absval(RecTypeC), MName, absval(ArgTypeC)) :−
   within((ExprID, calls((Rec, _), MName, _)), _, _),
   stype(Rec, RecType),
   subtypeeq(RecTypeC, RecType),
   meth(RecType, MName, meth(_, MName, ArgType, _)),
   subtypeeq(ArgTypeC, ArgType),
   addshadow((Time, ExprID)).
```

This rule uses the abstractions defined in Fig. 11 in order to create the virtual trace. Timestamps are represented by a pair `(Time,ExprID)`, whereby `Time` is either the constant `now` or it is unbound. To achieve this, we fix the `now` predicate to the definition `now((now,_))`. This means that all queries getting the timestamp via the `now/1` predicate will have their timestamp in the abstraction fixed to the constant `now`. All other queries will have an unbound variable in the first position of the timestamp; an unbound variable in the first position denotes a query that might refer to the past.

Instead of values, the execution trace uses types of the form `absval(SomeType)`. All method calls (found in the AST via `within`) imply a corresponding `calls` predicate, whereby the information from the static type system (`stype/2` predicate) is used in order to infer the type of the receiver. Subtyping is taken into account by corresponding `subtypeeq` constraints.

Of particular interest is the `addshadow` part of the rule. This is a special predicate that is intercepted by our static analysis. Whenever an `addshadow` goal is encountered, the interpreter adds the corresponding join point shadow (i.e., `ExprID`) and its lifetime (`Time`) to a list of shadows for the pointcut that is currently analyzed.

The definition of the virtual store is relatively straightforward: It defines the `store` and `classOf` predicate in terms of types instead of values. The situation becomes a bit complicated by taking subtype polymorphism into account. We deal with this by letting `store` range over all possible combinations of types in an object – we ignored performance and favored simplicity in our prototype analysis. The definition of the virtual store is also available in the appendix (Fig. 16).

```
subtypeeq(D, display ),
shadows(
  ( set(P, F, _), get(T1, _, P, F, _), calls (T2, _, absval(D), draw, _),
    cflow(T1, T2), instanceof(P, figureElement )), ,S)
```

Fig. 12. Query for shadows of `cflow` pointcut (line 17, Fig. 8)

```
class Point extends FigureElement {
  void draw(Display d) {
    if( this.enabled ) d. paintPoint ( this.x , this.y );
  }
}
class Line extends FigureElement {
  void draw(Display d) {
    this .foo( true );
    if( this.enabled ) d. paintLine( this.start , this.end );
  }
}
class Main extends Object {
  ...
  void writeSomething() {
    this .p2.y := false ; this .p1.x := true ;
    this .p1.enabled := false ;
  }
  void main() {
    this .p1 := new Point (); this .p2 := new Point ();
    this .p2.enabled := true ;
    this .l1 := new Line(); this .l1 . start := this .p1;
    this .l1 .end := new point;
    this .d := new Display (); this .d.f1 := this .l1 ;
    this .d.f2 := this .p2; this .du = new Displayupdate ();
    this .du.d := this .d;
    deploy( this .du) { this.d.drawAll() ; this .writeSomething (); }
  }
}
```

Fig. 13. The result of abstract interpretation

Our pointcut interpreter is implemented as a meta-interpreter in Prolog. Meta-interpreters are a common technique for abstract interpretation of logic programs [7]. Our meta-interpreter is basically the so-called *vanilla* meta-interpreter [36–Program 17.5] extended by a loop detection mechanism and an additional parameter that collects shadows. In order to invoke the pointcut interpreter we first have to substitute the dynamic values in the pointcut expressions with their static abstraction. By evaluating the pointcuts over the abstracted domains with our pointcut interpreter we basically perform a constant propagation analysis through the control flow of a pointcut. The code of the meta-interpreter is available in the appendix (`shadows` predicate in Fig. 16). We do not want to discuss its implementation here in detail because it uses some very Prolog-specific mechanisms.

For illustrating the abstract interpretation process, the query to compute the shadows for the *cflow* pointcut from line 17, Fig. 8 is shown in Fig. 12. The program expressions inside the pointcut (e.g., the `@this.d` expression in Fig. 8) are replaced by an abstract value that is constrained by its static type via a `subtype` constraint.

The meta-interpreter computes *all* solutions of the query on top of the virtual execution trace and virtual store (thereby collecting shadows triggered by `addshadow` goals). It is important that *all* solutions are computed such that the back-tracking evaluation of queries covers all possible evaluation scenarios at runtime. The abstract values (i.e., types) returned by the pointcut interpreter are also used to get a bound for the static type of pointcut variables, which is then used to type-check advice bodies.

Fig. 13 illustrates the result of computing the shadows for the aforementioned `cflow` pointcut (Fig. 8) in terms of the code from Fig. 4 and a sample main class. The shadows identified by the pointcut-interpreter are framed in Fig. 13. If the lifetime of the produced facts is `indefinite` (i.e., constant `now` has not been found in the timestamp, the expressions are also underlined, otherwise the lifetime is `immediate`.

For example, the call to `draw` and the field reads are marked as "indefinite lifetime" because they could be relevant as past events in the evaluation of the `get` goal in line 17 of Fig. 8. The lifetime of the field assignments is marked as `immediate` because they are only relevant for this query if they are the current `now` event.

5.1 Results and Limitations

The results of the static analysis are directly used in our interpreter in that Prolog facts/queries are only evaluated at marked shadows. Also, events for shadows that are marked with lifetime `immediate` are discarded immediately after the evaluation of the corresponding query. Our interpreter can be run both with and without this optimization. The performance gain depends directly on the relation between marked shadows and unmarked shadows. The example in Fig. 8 runs approximately 4 times faster with the abstract interpretation optimization turned on. In a different example, where the percentage of marked shadows to unmarked shadows is smaller, the program runs 300 times faster. This result is not surprising because extending the database and evaluating queries is very expensive, but it indicates that it is possible to have a very expressive pointcut language that is expensive only if pointcuts are used that cannot be projected on a small set of shadows.

Our analysis technique still has several important limitations, though. First, the analysis itself, as it is presented here, is very slow and would not scale to real systems. It is also hard to guarantee termination of the static analysis in all cases; a typical problem of static analysis by meta-circular interpreters [7]. Our primary goal was to show the feasibility of a static analysis only, so we favored simplicity over performance and completeness. We think that our analysis can be embedded into the conceptual framework described by Codish and Søndergaard [7]. They use a different meta-interpreter, a so-called "bottom-up" interpreter, that has better performance properties and is guaranteed to terminate. This is part of our future work.

Another limitation is in the existence of the `indefinite` lifetime because this means that such facts will never be removed from the database. A more fine-grained analysis that computes lifetimes of the kind "this fact can be removed after some event happened" would be desirable in order to remove this limitation. It is of course easy to construct queries that will inherently require indefinite storage of previous events, but in these cases the static analysis could be used to detect those queries and signal an error if the memory requirements cannot be restricted in a reasonable way.

How do we get from our prototype to an efficient implementation in a compiled language? Besides the limitations mentioned above, our representation of the store is not easy to implement efficiently. A trivial solution is to drop the store model from the pointcut language - there are no conceptual dependencies of our approach on the existence of the store (or any other) model. An alternative would be a database-like organization of the store, which is actually part of our future work.

In order to make the evaluation of the queries itself more efficient, we plan to use partial evaluation techniques such as Logen [26] to reduce dynamic pointcut evaluation to a minimum and to inline the remaining dynamic checks at the computed join point shadows.

6 Related Work

6.1 Pointcut Languages

Gybels' and Brichau's proposal [15] is related in several ways. Similar to our approach, they use logic programming and unification for matching pointcuts. The insertion of dynamic context into a pointcut similar to our `@expr` expressions is possible by means of *linguistic symbiosis* [3]. As in our approach, pointcuts can be made reusable by means of logic rules. The possibility of user-defined pointcut predicates or pointcut libraries is not discussed in [15], but this is no conceptual limitation.

The most important difference to our approach is the data model upon which pointcuts can be expressed. In their approach, the data model consists of a representation of the current join point, syntax tree, and some special object reifying predicates. It is hence not possible to encode queries that refer to the execution history or need access to data from the store. An efficient implementation by computing shadows of pointcuts is also discussed but the addition of the whole execution trace as in our case makes the problem much harder. Other works from the same group [19, 37, 4] also use logic meta-programming but consider only the static syntax of the program as data model.

LogicAJ [32] is an extension to AspectJ that uses logic variables and unification instead of wildcards in order to make pointcuts more expressive. The data model upon which the pointcuts operate is unchanged, though.

We have developed an extension of Alpha with which it is possible to refer to future events [24]. Due to several limitations of the implementation, this extension should be seen as an experiment to explore the limits of pointcuts and not as a proposal for a practical programming language.

Walker et al have developed an extension to AspectJ for expressing temporal relations between join points [38]. These temporal relations can be expressed via context-free grammars. The program trace is then "parsed" by an automaton for the grammar. Information about the history of the execution is stored in the state of these automata, which is an effective solution to reduce the amount of data that has to be stored. This approach would not be directly applicable to our model because our pointcut language is more powerful than context-free grammars.

Douence et al have proposed a special pattern matching language for execution traces based on Haskell [11]. Other models besides the execution trace are not covered. Many of the issues presented in this paper (integration into the language, context passing, efficient implementation) are not discussed.

Josh [5] is an AspectJ-like language with an extensible pointcut mechanism, built on top of Javassist [6]. Josh does not support declarative pointcut specifications. Rather, new PCDs in Josh are implemented as imperative meta-programs on the abstract syntax tree using the Javassist library. Josh basically suffers from the problems of an imperative meta-programming approach, especially with respect to the composability of the PCDs implemented as meta-programs.

Eichberg et al discussed the usage of the functional query language XQuery as an extensible pointcut language [13]. The data model in this approach is an XML representation of the abstract syntax tree. Due to functional abstraction and the module system of XQuery, it is possible to organize reusable pointcuts in libraries. Other data models or the integration into a programming language are not discussed.

Sakurei et al. [34] propose a design to extend AspectJ with object-specific aspects and pointcuts. Our `deploy` statement can be used with a similar effect as the `associate` statement in this approach. Since runtime values can be used directly in our pointcut language, arbitrary object-specific constraints can be expressed and not just those that are defined in a `perObject` clause. On the other hand, the proposal in [34] has a more a efficient implementation if many instances of the same object-specific aspect are active simultaneously.

6.2 Weaving and Static Analysis

Hilsdale and Hugunin have described the weaving mechanism in AspectJ [17]. The AspectJ weaver also computes shadows for dynamic pointcuts. However, AspectJ has only a fixed, predefined set of pointcut operators, hence it is easier to compute the set of join point shadows statically. Due to the structure of pointcuts, only certain dynamic checks that look at the class of objects or operate on special stacks (for `cflow`), are required, such that these dynamic checks can be directly woven into the code.

A more semantics-based compilation model, based on a simplified model of AspectJ, can be found in [28]. Using partial evaluation, their model can explain several issues in the compilation processes, including how to find places in program text to insert aspect code and how to remove unnecessary run-time checks. Sereni and de Moor describe a static analysis technique [35] for an even more simplified version of the AspectJ pointcut language that allows a more efficient implementation of some pointcuts than the implementation proposed in [28].

Douence et al presented an analysis technique for detecting interactions between aspects [10]. This is complementary to our static analysis, because we simply assume a global ordering among aspects and concentrate on computing *shadows* of single pointcuts. Nevertheless, our abstract interpretation implies a primitive interaction analysis for free, namely in that it becomes trivial to detect whether two pointcuts have intersecting shadows. However, this is not in the focus of our work.

Codish and Søndergaard [7] describe the usage of meta-interpreters for different abstract interpretations of Prolog code. In contrast to their "bottom-up" approach, we use a conventional top-down meta-interpreter with loop-detection. We are not aware of other works that use abstract interpretation for computing join point shadows.

6.3 Aspects and Modularity

Lopes et al. [27] motivate and speculate about future ”more naturalistic” referencing mechanisms inspired by natural languages, such e.g., *”those (data) read in previous sentence”*, or even *”in this last operation”*. By means of a simple example they illustrate how referencing mechanisms of current programming languages force programmers to circumscribe their intentions in terms of operational details of the underlying machine. They argue that while pointcuts in AOP languages go one important step further in supporting more powerful referencing they do not go far enough, e.g., in that they lack means of temporal referencing. The prototype we presented in this paper provides a very good basis to experiment with programming models that support more naturalistic referencing mechanisms as those envisaged in [27]. Our prototype can be extended to collect more and different kinds of information about the program to support more ”types of referencing”.

Aldrich [1] proposes module constructs that export pointcuts as part of the module specification. The rationale for this is the lack of modular reasoning if pointcuts depend on implementation details of a module. He shows that the implementation of such modules can be changed without affecting the consistency of the whole system. On the other hand, this approach is also a serious restriction to the programming model because 1) pointcuts of a module have to be anticipated in its design, 2) the existence of these pointcuts in the interface establishes an implicit coupling to the aspects that use the pointcut, and 3) if pointcuts go across modules (as is inherent for crosscutting concerns), the specification of the pointcut interfaces themselves becomes a crosscutting concern. Our approach also tackles this problem, but with very different means, namely by making the pointcut language more powerful, such that pointcut specifications can be made more robust and less dependent on implementation details. On the other hand, we can give no static guarantees because we cannot enforce implementation-independent pointcuts.

6.4 Information Engineering in Program Models

There are also some interesting related works outside the domain of programming language design. Efficient ways to manage and retrieve dynamic data about the execution of a program have been discussed by De Pauw et al [9]. Both the works by Lange and Nakamura [25] and by Richner and Ducasse [33] discuss the design of a static and a dynamic model of the program semantics as well as the use of logic rules to collect and combine information from these models in order to improve program understanding and program visualizations. Abstraction mechanisms to select interesting events in the execution of a program are also used in the domain of *debugging*, for example in the work of Jahier and Ducasse [18]. Reiss and Renieris have developed a framework for processing execution traces by reducing the amount of data as it is collected through mechanisms such as automata or context-free grammars [31]. These techniques may be helpful for us in order to further reduce the amount of collected data.

7 Summary and Future Work

In this paper we have presented an analysis which shows that current pointcut languages support localization of crosscutting concerns but have some problems with respect to information hiding. And we have described a new pointcut language in the form of logic queries over different models of the program semantics. Together with the abstraction facilities of logic programming, it becomes possible to raise the abstraction level of pointcuts and hence increase the software quality of aspect-oriented code. We have also presented a static analysis technique that can be the starting point of an efficient implementation.

Our future work will concentrate on the embedding of our pointcut language into a real compiled programming language and on further research in efficient implementation techniques that eliminate the limitations of our current analysis.

Acknowledgments

We would like to thank Gregor Kiczales, Michael Haupt and Michael Eichberg for comments on drafts of this paper.

This work is partly supported by the European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe).

References

[1] J. Aldrich. Open modules: Modular reasoning about advice. In *ECOOP'05: European Conference on Object-Oriented Programming*. Springer LNCS, 2005.

[2] Alpha project. http://www.st.informatik.tu-darmstadt.de/pages/projects/alpha/.

[3] J. Brichau, K. Gybels, and R. Wuyts. Towards a linguistic symbiosis of an object-oriented and a logic programming language. In *Proceedings of the Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL 2002)*, 2002.

[4] J. Brichau, K. Mens, and K. D. Volder. Building composable aspect-specific languages with logic metaprogramming. In *Generative Programming and Component Engineering (GPCE'02)*. Springer LNCS, 2002.

[5] S. Chiba and K. Nakagawa. Josh: An Open AspectJ-like Language. In *Proceedings of AOSD 2004*, Lancaster, England, 2004. ACM Press.

[6] S. Chiba and M. Nishizawa. An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. In *Proceedings of GPCE '03*, Lecture Notes in Computer Science, pages 364–376. Springer, 2003.

[7] M. Codish and H. Søndergaard. Meta-circular abstract interpretation in Prolog. In T. Mogensen, D. Schmidt, and I. H. Sudburough, editors, *The Essence of Computation: Complexity, Analysis, Transformation*, LNCS 2566. Springer, 2002.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages*. ACM Press, 1977.

[9] W. De Pauw, D. Kimelman, and J. M. Vlissides. Modeling object-oriented program execution. In *ECOOP '94: Proceedings of the 8th European Conference on Object-Oriented Programming*, pages 163–182, London, UK, 1994. Springer-Verlag.

[10] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *LNCS*. Springer-Verlag, 2002.

[11] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proc. of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, volume 2192 of *LNCS*. Springer-Verlag, 2001.

[12] S. Drossoupolou. Lecture notes on the L2 calculus. http://www.doc.ic.ac.uk/~scd/Teaching/L1L2.pdf.

[13] M. Eichberg, M. Mezini, and K. Ostermann. Pointcuts as functional queries. In *Second ASIAN Symposium on Programming Languages and Systems (APLAS)*. LNCS, 2004.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[15] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 60–69. ACM Press, 2003.

[16] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings OOPSLA '02. ACM SIGPLAN Notices 37(11)*, pages 161–173. ACM, 2002.

[17] E. Hilsdale and J. Hugunin. Advice Weaving in AspectJ. In *Proc. of AOSD'04*. ACM Press, 2004.

[18] E. Jahier and M. Ducasse. Generic program monitoring by trace analysis. In *Theory and Practice of Logic Programming Journal*, volume 2(4-5). Cambridge University Press, 2002.

[19] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *Proceedings of AOSD'03*. ACM Press, 2003.

[20] G. Kiczales. Keynote talk at AOSD '03, 2003.

[21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP '01*, 2001.

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings ECOOP'97*, LNCS 1241, pages 220–242. Springer, 1997.

[23] G. Kizcales and M. Mezini. Aspect-oriented programming and modular reasoning. In *Proceedings International Conference on Software Engineering (ICSE) '05*. ACM, 2005.

[24] K. Klose and K. Ostermann. Back to the future: Pointcuts as predicates over traces. In *Workshop on Foundations of Aspect-Oriented Languages (FOAL) at AOSD'05*, 2005.

[25] D. B. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 342–357, New York, NY, USA, 1995. ACM Press.

[26] M. Leuschel, J. Jørgensen, W. Vanhoof, and M. Bruynooghe. Offline specialisation in Prolog using a hand-written compiler generator. In *Theory and Practice of Logic Programming*, volume 4, pages 139–191, 2004.

[27] C. V. Lopes, P. Dourish, D. H. Lorenz, and K. Lieberherr. Beyond AOP: Toward naturalistic programming. In *Proceedings Onward! Track at OOPSLA'03*, Anaheim, 2003. ACM Press.

[28] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Proceedings of Compiler Construction (CC2003), LNCS 2622*. Springer, 2003.

[29] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proceedings Conference on Aspect-Oriented Software Development (AOSD) '03*, pages 90–99. ACM, 2003.

[30] D. L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, 1972.

[31] S. P. Reiss and M. Renieris. Encoding program executions. In *International Conference on Software Engineering*, Toronto, Ontario, Canada, 2001. IEEE.

[32] T. Rho and G. Kniesel. Uniform genericity for aspect languages. Technical Report IAI-TR-2004-4, Computer Science Department III, University of Bonn, Dec 2004.

[33] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, Washington, DC, USA, 1999. IEEE Computer Society.

[34] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In *Proc. of AOSD'04*. ACM Press, 2004.

[35] D. Sereni and O. de Moor. Static analysis of aspects. In *Proceedings of AOSD'03*. ACM, 2003.

[36] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1994.

[37] K. D. Volder and T. D'Hondt. Aspect-Oriented Logic Meta Programming. In *Conf. Meta-Level Architectures and Reflection, LNCS 1616*. Springer, 1999.

[38] R. J. Walker and K. Viggers. Implementing protocols via declarative event patterns. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-12)*, 2004.

A Appendix

Format	Example

prog(
  [class(ClassName, SuperClass,
   [field(FieldType, FieldName), ...
   [meth(RetType, MethName, ArgType, Expr), ...],
   [advice(before, Expr), ...]
  ] ...
)
where Expr has the form:
(ExprID, if(IfExpr, ThenExpr, ElseExpr)
(ExprID, get(ReceiverExpr, FieldName)
(ExprID, seq(Expr1, Expr2)
 ...
	prog(
	  [class(point, figureElement,
	    [field(bool, xx), field(bool, yy),
	      field(bool, enabled)],
	    [meth(bool, draw, display),
	     ('9:5', if(('9:13', get(('9:8',this),enabled)),
	       ('10:7', '10:15', seq(
	         ('10:13', get(('10:8', this), xx)),
	         ('10:22', get(('10:17', this), yy)))),
	       ('11:10', true))))],
	    []%no advice
	   )%class point
	  ]%classes
	). %prog

stype(ExprID, Type)	stype('2:26', bool)

new(ID, ExprID, ClassName, Obj)
calls(ID, ExprID, Receiver, MethodName, Arg)
set(ID, ExprID, Receiver, FieldName, Value)
get(ID, ExprID, Receiver, FieldName)
deploy(ID, ExprID, Obj)
endcall(ID, CallID, ReturnValue)
 	
pred(ID1, ID2) % event ID1 happened
   % immediately before event ID2
before(ID1, ID2) % transitive hull of pred
now(ID) % gives the current event ID
	calls(3, '53:5', iota1, setP1, iota2)
	set(4, '66:14', iota1, p1, iota2)
	endcall(5, 3, false)

store(Obj, FieldName, Value)
classof(Obj, ClassName)
	classof(iota2, point)
	store(iota2, enabled, false)
	store(iota2, yy, false)
	store(iota2, xx, true)

Fig. 14. Format of the four program models (AST, static typing, execution trace, object store) available for pointcuts in Alpha

```
% abbrevations if only interested in current event
new(ClassName, Obj) :−
  now(ID), new(ID, _, ClassName, Obj). ∗
calls (Receiver, Method, Arg) :−
  now(ID), calls (ID, _, Receiver, Method, Arg).
set (Receiver, Field, Val) :−
  now(ID), set (ID, _, Receiver, Field, Val).
get(Receiver, Field) :−
  now(ID), get(ID, _, Receiver, Field).
deploy(Receiver) :−
  now(ID), deploy(ID, _, deploy(Receiver)).

% is ID0 in the control flow of ID1?
cflow(ID0, ID1) :−
   calls (ID1, _, _, _, _), before(ID1, ID0),
   endcall (ID2, _, ID1, _), before(ID0, ID2).
cflow(ID0, ID1) :−
   calls (ID1, _, _, _, _),  before(ID1, ID0),
   \+ encall (_, _, ID1, _).

% is Obj2 reachable from Obj1?
reachable (Obj1,Obj2) :− reachablevia (Obj1,Obj2 ,[]).
reachablevia (Obj1,Obj2,_) :− store (Obj1, _, Obj2).
reachablevia (Obj1,Obj2,Via) :−
   store (Obj1, _, Obj3), \+ member(Obj1,Via),
   reachablevia (Obj3, Obj2, [Obj3|Via]).

% convenient access of AST
class (Name, CDef) :−
  prog(CDefs), member(CDef, CDefs),
  CDef = class (Name, _, _, _, _).
meth(CName, MName, MDef) :−
   class (CName, class(_, _, _, MDefs, _)),
  member(MDef, MDefs), MDef = meth(_,
MName, _, _).
field (CName, FName, FDef) :−
   class (CName, class(_, _, FDefs, _, _)),
  member(FDef, FDefs), FDef = field (_, FName).
```

```
within ((ExprID, Expr), CName, MName,) :−
  meth(CName, MName, meth(_, _, _, Body)),
  subExpr(Body, (ExprID, Expr)).
subExpr(E, E).
subExpr(X, E) :−
  X =.. [_| List ], member(E1, List), subExpr(E1, E).

% static subtype/subclass relation
directsubtype (C1, C2) :−
   class (C1, class (_, C2, _, _, _)).
subtypeeq(bool, bool).
subtypeeq(C, C) :− class (C, _).
subtypeeq(C1, C2) :−
   directsubtype (C1, C3), subtypeeq(C3, C2).

% add subtyping to static and dynamic types
hastype(ExprID, C) :−
  stype(ExprID, D), subtypeeq(D, C).
instanceof (Obj, C) :−
   classof (Obj, D), subtypeeq(D, C).

%predicted control flow
pcflow(CName, MName, E) :−
  pcflow1(CName, MName, E, []).
pcflow1(CName, MName, E, _) :−
  within(E, CName, MName).
pcflow1(CName, MName, E, V) :−
  within ((_, calls ((RecID, _), MName1, _)), CName, MName),
  stype(RecID, CName2),
  (subtypeeq(CName1, CName2); subtypeeq(CName2, CName1)),
  meth(CName1, MName1,_),
  \+ member((CName1, MName1), V),
  pcflow1(CName1, MName1, E, [(CName1,MName1)|V]). ∗

%finding the most recent of an event pattern X
mostRecent(ID,X) :−
  bagof(ID,X,IDs), maxlist (IDs,ID).
```

Fig. 15. Standard pointcut library

```
% virtual store
store ( absval (CName), Field, absval (TypeC)) :−
        subtypeeq(CName, CSuper),
         field (CSuper, Field , field (Type, _)),
        subtypeeq(TypeC, Type).
classof ( absval (CName),CName).

% virtual event trace
calls ((Time,ExprID), ExprID,
  absval (RecTypeC), MName, absval(ArgTypeC))
:−
  within ((ExprID, calls ((Rec, _), MName, _)), _, _),
  stype (Rec, RecType),
  subtypeeq(RecTypeC, RecType),
  meth(RecType, MName, meth(_, MName, ArgType, _)),
  subtypeeq(ArgTypeC, ArgType),
  addshadow((Time, ExprID)).
% similarly for set , get , new,deploy, endcall
```

```
now(now).
before(_, _).
pred(_, _).
% the meta−interpreter
shadows(true , [], _) :− !.
shadows((A,B), Shadows, Trail ) :−
   !, shadows(A, S1, Trail ), shadows(B, S2, Trail ),
  append(S1, S2, Shadows).
shadows(X, [], _) :−
   predicate_property (X, built_in ), !, X.
shadows(addshadow((Time,Token)), S, _) :−
  var(Time), !, S = [(Token, indefinite )].
shadows(addshadow((now,Token)), S, _) :−
   !, S = [(Token, immediate)].
shadows(A, S, Trail ) :−
   loop_detect (A, Trail ), !.
shadows(A, S0, Trail ) :−
  clause (A, B), shadows(B, S0, [A| Trail ]).
```

Fig. 16. Meta-interpreter for computing pointcut shadows

Sustainable System Infrastructure and Big Bang Evolution: Can Aspects Keep Pace?

Celina Gibbs, Chunjian Robin Liu, and Yvonne Coady

University of Victoria, British Columbia, Canada
{celinag, cliu, ycoady}@cs.uvic.ca

Abstract. Realistically, many rapidly evolving systems eventually require extensive restructuring in order to effectively support further evolution. Not surprisingly, these overhauls reverberate throughout the system. Though several studies have shown the benefits of aspect-oriented programming (AOP) from the point of view of the modularization and evolution of crosscutting concerns, the question remains as to how well aspects fare when the code that is crosscut undergoes extensive restructuring. That is, when evolution is a big bang, can aspects keep pace? The case study presented here considers several categories of aspects – design invariants, dynamic analysis tools, and domain specific design patterns – and shows the concrete ways in which aspects had positive, negative and neutral impact during the restructuring of the memory management subsystem of a virtual machine. Compared with best efforts in a hierarchical decomposition coupled with a preprocessor, aspects fared better than the original implementation in two out of four aspects, and no worse in the remaining two aspects.

1 Introduction

Legacy systems eventually face upheavals in system structure. The dawn of new system structure, marked by improved separation of concerns, is often preceded by a darkness in which the old structure must be torn down. This explosive, *big bang*, type of evolution forces simultaneous changes throughout the system. Though aspects have been shown to be effective as a locus of control for evolving crosscutting concerns, the fact that they rely on explicit external interaction implies that aspects could have negative impact under these extreme conditions – when the *code that is crosscut*, or the dominant decomposition – is undergoing structural reorganization.

Low level system infrastructures need to be fast yet flexible – traits that are commonly at odds with each other. To reconcile this tension, standard practices within this domain include preprocessor directives and system patch files. These mechanisms are de facto standard in part because they introduce no performance overhead, and in part because they provide at least rudimentary means to achieve better configurability and extensibility than traditional language constructs in C and Java. Though distasteful to many developers due to their lack of semantic levarage, they are a reality in today's system infrastructure software.

In order to test the sustainability of aspects in systems such as these, we conducted an experiment using the rapidly evolving Memory Management Toolkit (MMTk) [2], within the Jikes Research Virtual Machine (RVM) [15]. The RVM is a unique open source project in Java. A basis for almost 100 publications over the last 5 years and averaging several hundred CVS commits per month, the RVM is host to most of today's state-of-the-art java virtual machine technology. Its code-base affords researchers the opportunity to experiment with a variety of design and implementation alternatives within an otherwise stable and consistently well maintained infrastructure. Specifically, MMTk is a framework used within the RVM designed to support research in new garbage collection (GC) strategies. MMTk provides a marked departure from traditional monolithic implementations by being both more modular and efficient than its predecessors [2].

In the experiment described here, we compared the evolution of the original system (MMTk) versus one where we introduced aspects (MMTk_{ao}). The aspects included representatives from three different categories: design invariants, dynamic analysis tools [29], and domain specific design patterns [2]. In the original implementation, the Jikes RVM relies on preprocessor directives along with hierarchical decomposition to achieve a highly efficient, configurable system. In order to determine if aspects could keep pace with evolution in a system based on this composition, we considered evolutionary restructuring tasks over an intensive change period of 10 months (version 2.3.3 from January – October 2004). In total, 12 significant restructurings across four aspects were considered in the study, where a single restructuring often had impact on multiple aspects. The aspects were programmed using AspectJ [19, 14].

With respect to the impact of the 12 restructuring tasks, the results show that evolution of aspects fared better than the original implementation in two of four aspects, and no worse in the remaining aspects. Results from the DaCapo Benchmark for GC [5] show the system sustained a worst-case performance penalty just under an average of 10% as a result of one fine-grained aspect with of millions of advice invocations. This study demonstrates the ways in which existing mechanisms for AOP can support sustainable system infrastructure relative to their preprocessed/hierarchical counterparts in the somewhat inhospitable domain of system infrastructure software. Further, the resulting system structure is arguably better suited for future trends in evolution.

This paper is organized as follows. After an overview of related work, the implementation and evolution of MMTk is described (section 2), followed by a description of the same evolution applied to MMTk_{ao} (section 3). In terms of analysis, the evaluation of sustainability considers each category of aspect, limitations of the study, and preliminary performance considerations (section 4).

1.1 Related Work

AOP provides linguistic support aimed at improving modularity. The software engineering community has repeatedly demonstrated that modularity plays a key role in determining the cost of change [24, 6, 21]. Unfortunately, structural boundaries tend to decay over time due to increasing dependencies between modules [27, 20]. Structural deficiency results in the need for non-local changes that require considerable effort associated with non-local reasoning [32, 25]. Evolution thus becomes impaired over time, as modularity becomes compromised.

Original case studies involving AOP implementations generally reveal qualities associated with improved separation of concerns [22, 17, 12]. More recently, aspects have been associated with higher quality code refactorings [28, 10], adaptability in middleware [4, 33, 7], configurability in real time systems [30], and autonomic computing in OS kernels [8]. In terms of evolution, Banniassad's results show that the lack of adequate separation of concerns can stand as an obstacle to evolution [1], while Walker's work demonstrates that explicit separation can have mixed results in terms of understanding and evolving systems [31]. Rashid's study of evolution in database systems [26], and our own previous work in operating systems [3], have uncovered some of the benefits of aspects in terms of evolution.

The case study presented here evaluates aspects in the context of a particularly intense evolutionary scenario. The rapid refactoring of the memory management subsystem of the Jikes RVM provides a rich testbed for assessing sustainability in terms of modularity. In particular, the fact that the aspects needed to change in response to changes in the code they crosscut provided an acid test for aspects. That is, we were able to better establish, (1) the value of the internal structure of the aspects alongside, (2) the changes that occurred to the external interaction explicitly defined by pointcuts. This allowed us to view evolution of the dependencies between the concerns from the perspective of the semantics of the woven system as a whole.

2 Evolution of MMTk

The evolution of MMTk reveals a move towards a more portable and manageable subsystem. Portability is necessary as MMTk can now be used in other systems requiring garbage collection, such as the Glasgow Haskell Compiler [13] and OVM [23]. Manageability is necessary as MMTk serves as platform for developers to experiment with new GC strategies. As a result of these two main motivating factors, key evolutionary steps of MMTk include: (1) new adherence to a strict interface for portability, and (2) new hierarchical decomposition and migration of code within (sub)packages to better separate concerns. A high level overview of the extensive nature of this evolution over a 10 month period is provided by Table 1, with more detailed accounting of 12 major restructurings in Table 2.

Table 1. Evolution in MMTk: old and new class sturucture overview

TOP LEVEL PACKAGE	# OF OLD CLASSES	EVOLVED CLASSES
GCspy	14	moved out of MMTk
Plan	22	15 (new hierarchy)
Policy	10	15 (new hierarchy)
Utility	63	14 (+5 new subpackages)
VMInterface	17	redistributed in MMTk

Table 2. Detailed evolution of 12 restructuing tasks of MMTk over 10 months

BETTER SEPARATION OF MMTK FROM JIKES	BETTER SEPARATION OF CONCERNS WITHIN MMTK	GENERAL EVOLUTION AND MAINTENANCE
1.1) Eliminated MMTk and VM_Magic interaction	2.1) Restructured plan package	3.1) One class changed to implement synchronization interface
1.2) Eliminated utility classes	2.2) Restructured policy package	3.2) Eliminated assertion or failure methods
1.3) Relocated and renamed VM_Address class	2.3) Restructured utility package	3.3) Introduced new classes
1.4) Relocated and renamed synchronization interfaces	2.4) Redistristributed and eliminated VM_Interface	3.4) Introduced calls to assertion or failure methods

The 12 restructurings are significant as they form the basis of the experiment for evolution in this study. The columns represent different categories of restructuring. On the left, changes *1.1-1.4* were necessary to make MMTk more portable, and to make its separation from Jikes specific code more hygienic. In the middle, changes *2.1-2.4* supported better separation of concerns within MMTk, making it easier for developers to experiment with GC. The final column, changes *3.1-3.4* represent more generic tasks associated with maintenance and evolution.

2.1 Inheritance and Preprocessing: Structured, Efficient, and Configurable

MMTk supports the selection of one of many different GC strategies or *plans.* In the old implementation there was a package for each GC plan, which contained the implementation of core features in the *Plan* class. The specific Plan class to be include in a particular build of the system is determined by a command line option. In the evolved implementation, these subdirectories were eliminated and each GC strategy has its own distinctly named class (CopyMS.java, SemiSpace.java, etc.). The Plan class was migrated outside of MMTk, and made a subclass of each of these classes mutually exclusively, using preprocessor directives:

```
//-#if RVM_WITH_SEMI_SPACE
public class Plan extends SemiSpace implements Uninterruptible {
//-#elif RVM_WITH_SEMI_SPACE_GC_SPY
public class Plan extends SemiSpaceGCSpy implements Uninterruptible {
//-#elif RVM_WITH_COPY_MS
public class Plan extends CopyMS implements Uninterruptible {
…
(more plan classes)
//-#endif
```

The Plan class highlights another way in which plans in MMTk evolved, further leveraging hierarchical decomposition and this multiplexed approach. *Semi_Space_GC_Spy* uses a different class than the regular *Semi_Space* plan. This better separates instrumentation for the dynamic analysis tool, GCSpy, from the regular plan. *Semi_Space_GC_Spy* and *Semi_Space* are siblings under *Semi_Space_Base.* Shared code is in the base class, functions instrumented for GCSpy are in the *GC_Spy* child, and uninstrumented versions of those functions are in the regular class. Pre-evolution, this separation was not supported by a class hierarchy. Instead, there were two versions of Semi_Space plan, one with instrumentation and the other without. Figure 1 overviews these two implementations.

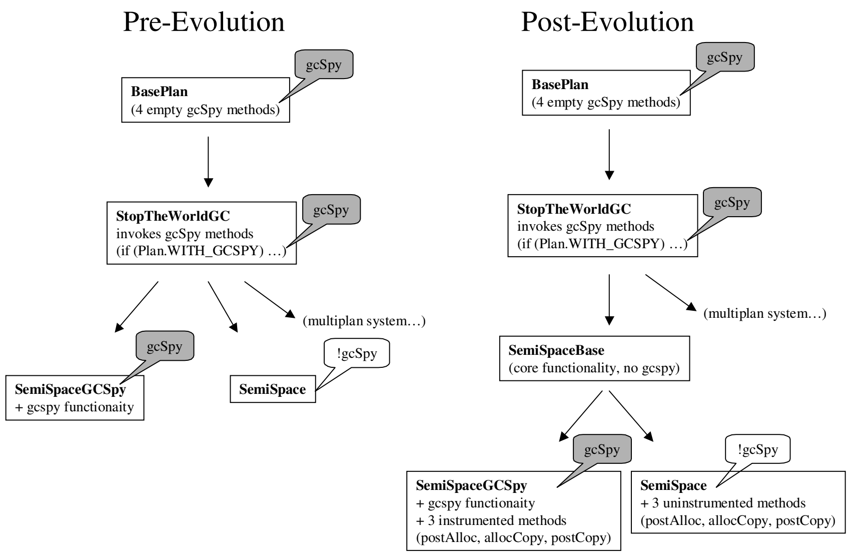

Pre-Evolution
Post-Evolution
gcSpy
BasePlan (4 empty gcSpy methods)
StopTheWorldGC invokes gcSpy methods (if (Plan.WITH_GCSPY) …)
(multiplan system…)
SemiSpaceGCSpy + gcspy functionaity
SemiSpace
!gcSpy
SemiSpaceBase (core functionality, no gcspy)
SemiSpaceGCSpy + gcspy functionaity + 3 instrumented methods (postAlloc, allocCopy, postCopy)
SemiSpace + 3 uninstrumented methods (postAlloc, allocCopy, postCopy)

Fig. 1. GCSpy in Pre and Post-evolution in MMTk

2.2 Empty Interfaces: When *Implements* Is a Lower Level Concern

One of the other interesting features of the Jikes system infrastructure is its lightweight leveraging of empty interfaces to flag concerns handled by lower level system code. For example, the majority of the classes in MMTk implement an interface called *VM_Uninterruptible (pre-evolution)* or *Uninterruptible (post-evolution).* Classes that implement this interface cannot be interrupted. The mechanism to supply the functionality for this concern however, is actually provided by a lower level of the system, outside of the implementation of the Jikes core.

3 Evolution in MMTk$_{ao}$

To build our aspect-oriented version of MMTk, called MMTk$_{ao}$, we took the pre-evolution version of MMTk, before the 10 month evolution period, and factored out aspects for the following four, diverse, crosscutting concerns:

Crosscutting concerns (CCC)	**Interacting Concerns (IC) (in terms of packages)**
DesignInvariants: VerifyAssertions Synchronization	MMTk.*
GCSpy	plan, policy, utility
Prepare/Release Protocol	plan

Detailed analysis of this original refactoring can be found in previous work [9]. The contribution of the work presented here focuses on the evolution of $\mathrm{MMTk}_{\mathrm{ao}}$, and the sustainability of these aspects during an intense period of system evolution. An overview of how the evolution of the interacting concerns (i.e., the code that is crosscut by these aspects) impacted each aspect is provided in the subsections that follow.

3.1 Design Invariant: Evolution of Assertions

Jikes RVM uses a boolean field, *VerifyAssertions*, as a global flag to enable assertion checking. The `VM` class, which originally was home for this flag, has a comment dictating the structure of this design invariant:

```
/* Note: code your assertion checks as
   "if (VM.VerifyAssertions) VM._assert(xxx);"  */
```

During evolution, the *VM_Interface* class was reorganized as several new classes, one of which was the *Assert* class, for better separation of this concern. Further refactoring eliminated a previously core method from this concern. The general structure of the design invariant remains across the evolution of the system, as shown in Table 3 and Figure 2. Since the system is so performance critical, the presence/absense of this code is significant, and evidence that it has been removed for performance but restored for correctness appears in comments in the CVS logs.

Table 3. *VerifyAssertions* across two versions of the *plan* and *policy* packages

OCCURRENCE OF...	PRE-EVOLUTION	POST-EVOLUTOIN
if (verify_assertions)	98 instances, 19 classes	81 instances, 21 classes
call to _assert(..)	75 instances, 25 classes	68 instances, 23 classes
call to sysFail/fail(..)	29 instances, 9 classes	31 instances, 14 classes
call to spaceFailure(..)	14 instances, 8 classes	no longer exists

In terms of accounting for changes inflicted upon this aspect by evolution of the code it crosscuts, this aspect deals with new/removed calls to assert/failure methods without requiring change (Table 2, 3.4), but still had a total of 8 points of change during the evolution. During the separation of MMTk from Jikes, global fields used by MMTk were moved to be within its boundaries (Table 2, 1.2). Cleaning of the

Assert class caused the removal of a method resulting in an obsolete pointcut/advice (Table 2, 2.4, and 3.2). A performance assessment of this aspect is presented in section 4.6.

PRE-EVOLUTION

```
privileged aspect VerifyingAssertions {

   pointcut GCstrategy():
       within(org.mmtk. *);

   pointcut asserting(boolean condition):
       call(void VM_Interface._assert(boolean))
       && args(condition)
       && GCstrategy();

   pointcut failing(String msg):
       call(void VM_Interface.sysFail(String))
       && args(msg)
       && GCstrategy();

   pointcut space_failure(VM_Address obj,
                          byte space,
                       String source):
       call(void Plan.spaceFailure(VM_Address,
                                 byte,
                                    String))
       && args(obj, space, source)
       && GCstrategy();

   void around(boolean b): asserting(b){
       if (VM_Interface.VerifyAssertions)
            proceed(b);
   }

   void around(String str): failing(str){
       if (VM_Interface.VerifyAssertions)
           proceed(str);
   }

   void around(VM_Address obj, byte space,
               String source):
         space_failure(obj,space,source) {
       if (VM_Interface.VerifyAssertions)
           proceed(obj,space,source);
   }
}
```

POST-EVOLUTION

```
privileged aspect VerifyingAssertions {

   pointcut GCstrategy():
       within(org.mmtk.*);

   pointcut asserting(boolean condition):
       call(void Assert._assert(boolean))
       && args(condition)
       && GCstrategy();

   pointcut failing(String msg):
       call(void Assert.fail(String))
       && args(msg)
       && GCstrategy();

   void around(boolean b): asserting(b){
       if (Assert.VerifyAssertions)
            proceed(b);
   }

   void around(String str): failing(str){
       if (Assert.VerifyAssertions)
           proceed(str);
   }

}
```

Fig. 2. *VerifyAssertions* Aspect

3.2 Design Invariant: Evolution of Synchronization

The aspect used for synchronization is very simple. It has a specific set of classes that do *not* need to implement a given interface that flags if the class is uninterruptible. As the majority of the classes in the subsystem cannot be interrupted, this list is of the exceptions to the rule. As previously mentioned, the interface itself is empty, and is used by a lower level concern.

During evolution, new classes added to the system all required the interface. The aspect deals with this correctly (Table 2, 3.3). In total, this aspect underwent two points of change in the evolution: the interface was renamed (Table 2, 1.4), and the BasePolicy class changed status (Table 2, 3.1), such that it required the interface (shown in a comment in Figure 3).

```
privileged aspect Synchronization {

  declare parents:
     (org.mmtk.* || com.ibm.JikesRVM.memoryMangers.mmInterface.*) &&
     !(*Header
       || org.mmtk.utility.alloc.AllocAdvice
       || org.mmtk.utility.TracingConstants
       || org.mmtk.utility.CallSite
   //  || org.mmtk.policy.BasePolicy
       || org.mmtk.vm.ScanStatics
       || org.mmtk.vm.Constants
       || com.ibm.JikesRVM.memoryManagers.mmInterface.MM_Constants
       || com.ibm.JikesRVM.memoryManagers.mmInterface.SynchronizationBarrier
       || com.ibm.JikesRVM.memoryManagers.mmInterface.VM_CollectorThread
       || com.ibm.JikesRVM.memoryManagers.mmInterface.VM_GCMapIteratorGroup
       || com.ibm.JikesRVM.memoryManagers.mmInterface.VM_Handshake)
     implements Uninterruptible;
}
```

Fig. 3. *Synchronization* Aspect

3.3 Evolution of GCSpy

The core implementation of GCSpy, a dynamic analysis tool for GC, involves two parts: (1) gathering of data before and after garbage collection, and (2) connecting to a GCSpy server and client-GUI for heap visualization. In order to instrument MMTk with this code, the configuration of the system with GCSpy requires existing methods to be instrumented, and new methods be added, as outlined in Table 4.

Table 4. Instrumentation of GCSpy

PACKAGE	CLASS	OCCURRENCES OF IF (GCSPY)	NEW METHODS
org.mmtk.plan	Plan (SemiSpace)	5	6
	BasePlan	0	5
	StopTheWorldGC	4	0
org.mmtk.utility	FreeListResource	1	0
	MonoToneVMResource	2	2
com.ibm.JikesRVM. memorymanagers mmInterface	MMInterface	2	0
	Total	*14*	*13*

In the original implementation, part of the configuration strategy for GCSpy involves checking a global flag, *if (VM_Interface.GCSPY)*, before invoking GCSpy functionality (Table 4). The flag is set using preprocessor directives, as follows:

```
public static final boolean GCSPY =
  //-#if RVM_WITH_GCSPY
  true;
  //-#else
  false;
  //-#endif
```

MainThread.java, part of the scheduler package, also uses the directive within its imports and to start the server:

```
//-#if RVM_WITH_GCSPY
import com.ibm.JikesRVM.memoryManagers.mmInterface.MM_Interface;
//-#endif

…
public void run () {
       //-#if RVM_WITH_GCSPY
       MM_Interface.startGCSpyServer();
       //-#endif
```

In *VM_BootRecord.java*, part of the system's runtime support package, 28 fields for GCSpy are declared when this directive is true and *VM_Syscall* contains the methods:

```
//-#if RVM_WITH_GCSPY
   // GCspy entry points
   public VM_Address gcspyDriverAddStreamIP;
   public VM_Address gcspyDriverEndOutputIP;
   …
   public VM_Address gcspySprintfIP;
//-#endif
```

VM_Syscall.java, has 28 syscall entry points:

```
//-#if RVM_WITH_GCSPY
public static VM_Address
gcspyDriverAddStream (VM_Address driver, int it) {
  return null; }

public static void
gcspyDriverEndOutput (VM_Address driver) {}

…
public static int
gcspySprintf (VM_Address str, VM_Address format,
              VM_Address value) { return 0; }
//-#endif
```

An assortment of several other classes, not in MMTk, use this directive to selectively import and introduce GCSpy functionality. The typical format for these classes is of the form: *#if RVM_WITH_GCSPY, <define method bodies>, #else <provide empty bodies>*. For example, in *ObjectMap.java:*

```
import com.ibm.JikesRVM.VM_SizeConstants;
import com.ibm.JikesRVM.VM_Uninterruptible;
import com.ibm.JikesRVM.VM_Address;

//-#if RVM_WITH_GCSPY
import org.mmtk.plan.Plan;
import org.mmtk.utility.Log;
…
//-#endif

/**
 * THIS CLASS IS NOT A GCSPY COMPONENT
 *
    …
*/

public class ObjectMap
  implements VM_SizeConstants, VM_Uninterruptible {

//-#if RVM_WITH_GCSPY
    private static final int LOG_PAGE_SIZE = 12;
    static final int PAGE_SIZE = 1<<LOG_PAGE_SIZE;
  …
  public ObjectMap() { }

  public final void boot() {
    objectMap_ = Util.malloc(OBJECTMAP_SIZE);
    VM_Memory.zero(objectMap_, OBJECTMAP_SIZE);
          …
  }
  …
//-#else
  public ObjectMap() {}
  public final void boot() {}
   …
//-#endif
}
```

Refactoring the portions of GCSpy handled by the preprocessor directives was straightforward. MMTk$_{ao}$ enjoys the added benefit of being able to plug/unplug at build time, instead of requiring the system to be first reconfigured and then rebuilt.

Within MMTk$_{ao}$, as opposed to the strategy of subclassing and introducing redundant code to allow for the GCSpy functionality (Section 2.1 and Figure 1), with minor refactoring of the plan, the aspect provides GCSpy functionality. Thus, the combination of global flags, preprocessor directives, and subclassing leveraged by the original implementation become unnecessary in MMTk$_{ao}$.

As GCSpy was the most thinly scattered concern considered in the study, involving many points in the execution of the system with almost a 1:1 ratio of pointcuts:advice, it was impacted by a majority of the restructuring in Table 2 (9/12 tasks, 1.1-1.3, 2.*, 3.1,2).

3.4 Evolution of Prepare/Release Protocol

To understand how each GC plan is composed one must understand their relationships with the various *policies* supplied in the RVM. Plan and policy are thus two key features of the dominant decomposition of the RVM. Each policy has its own allocation and collection strategies, drawn from basic allocation and collection mechanisms. The memory management GC plans in the RVM are composed of different combinations of these policies. Currently, there are eight different memory management plans available for download in the RVM.

Memory management in the RVM follows a simple algorithm of *prepare*, *process-all-work*, and *release* for collection. Since each of the GC strategies share the code base for *process-all-work*, the key differences between them are in the *prepare* and *release* phases. With a closer inspection of the original implementation a further breakdown of this design into *global prepare*, *local prepare*, *local release* and *global release* can be seen and represented as a simple finite state machine as illustrated in Figure 4. Each of these states is comprised of calls to the various policy mechanisms.

When the relationship between policy and plan is filtered out this way, a clear symmetry between the *prepare* and the *release* phases is uncovered. This symmetry is present in both the *local* and *global* scopes. Each of the policies involved in the *global prepare* are in turn involved in the *global release* and the same is true in the case of the *local* scope.

Blackburn et. al. detail the domain specific design patterns used in the implementation of MMTk, one being the *Prepare/Release* phases involved in garbage collection [2].

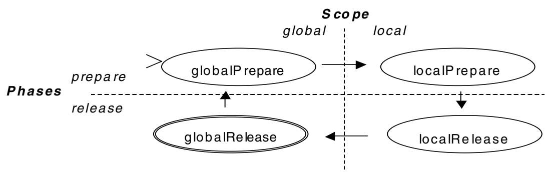


Fig. 4. Finite state machine for prepare/release protocol

```
//handles Prepare/Release of SemiSpace GC plan
privileged aspect PolicyAspect {

  private int state = 0;
  private final int GLOBAL_PREPARE = 0;
  private final int LOCAL_PREPARE =  1;
  private final int LOCAL_RELEASE =  2;
  private final int GLOBAL_RELEASE = 3;

  after(Plan p):target(p)
    && (execution(* Plan.globalPrepare(..))
    || execution(*
Plan.threadLocalPrepare(..))
    || execution(*
Plan.threadLocalRelease(..))
    || execution(* Plan.globalRelease(..))) {

        switch(state){
          case(GLOBAL_PREPARE):
                    CopySpace.prepare();
                    ImmortalSpace.prepare();
                    Plan.losSpace.prepare();
                    state++;
                    break;

          case(LOCAL_PREPARE):
                    p.los.prepare();
                    state++;
                    break;

          case(LOCAL_RELEASE):
                    p.los.release();
                    state++;
                    break;


          case(GLOBAL_RELEASE):
                    Plan.losSpace.release();
                    CopySpace.release();
                    ImmortalSpace.release();
                    state = GLOBAL_PREPARE;
                    break;
        }
   }
}
```

```
//handles Prepare/Release of MarkSweep
plan
privileged aspect PolicyAspect {

  private int state = 0;
  private final int GLOBAL_PREPARE = 0;
  private final int LOCAL_PREPARE =  1;
  private final int LOCAL_RELEASE =  2;
  private final int GLOBAL_RELEASE = 3;

  after(Plan p):target(p)
  //same pointcut as SemiSpace


{
        switch(state){
          case(GLOBAL_PREPARE):

          Plan.msSpace.prepare();
                    Immortal-
Space.prepare();

          Plan.losSpace.prepare();
                    state++;
                    break;

          case(LOCAL_PREPARE):
                    p.ms.prepare();
                    p.los.prepare();
                    state++;
                    break;

          case(LOCAL_RELEASE):
                    p.ms.release();
                    p.los.release();
                    state++;
                    break;

          case(GLOBAL_RELEASE):

          Plan.losSpace.release();

          Plan.msSpace.release();
                    Immortal-
Space.release();
                    state =
GLOBAL_PREPARE;
                    break;
        }
    }
}
```

Fig. 5. Pre-evolution version of SemiSpace and MarkSweep aspects

The Prepare/Release aspects for *SemiSpace* and *MarkSweep* GC plans are shown in Figure 5. By looking at the plans in such close proximity, the differences in policy mechanisms employed by the two plans are evident. This same representation is scalable to all plans, providing developers of new plans a clear view of current implementations and a well defined, staged-protocol to follow in the development of new plans.

Pre and post-evolution versions of these plans differ in their pointcuts as result of the combination of hierarchical restructuring and preprocessor directives described in Section 2.1 with respect to Plans (Table 2, 2.1). As a result, the pointcuts move from the generic use of Plan.<*method*>, to the specific use of *SemiSpace.<method>* and *MarkSweep.<method>*.

4 Sustainability Analysis

Table 5 summarizes the findings of this experiment. For each of the 12 restructurings, each of the aspects was forced to change by virtue of the fact that a concern that they interacted with (an interacting concern, *IC*) had changed. That is, while the dominant decomposition of the system is evolving, the core of the aspect's functionality is staying the same – only the explicit interaction must be redefined. Thus,

Table 5. Summary of changes: required change(Δ), change automatically captured (AC)

	Aspect / Corresponding IC(s) / Evolutionary Change	GCSpy	IC	Synchronization	IC	Verify Assertions	IC	Prepare/Release	IC
Separation of MMTk	1.1) Eliminated MMTk and VM_Magic interaction	Δ	Δ						
	1.2) Eliminated utility classes	Δ	Δ						
	1.3) Relocated and renamed VM_Address class	Δ	Δ						
	1.4) Relocated and renamed synchronization interfaces	Δ	Δ	Δ	Δ				
Separation of concerns win	2.1) Restructured plan package	Δ	Δ					Δ	Δ
	2.2) Restructured policy package	AC	Δ					AC	Δ
	2.3) Restructured utility package	AC	Δ						
	2.4) Redistributed and eliminated VM_Interface class	Δ	Δ			Δ	Δ		
Evolution/ Maintenance	3.1) Class changed to implement synchronization interface			Δ	AC				
	3.2) Eliminated assertion or failure functions					Δ	Δ		
	3.3) Introduced new classes	Δ	Δ	AC	Δ			AC	Δ
	3.4) Introduced new assertion/failure calls					AC	Δ		

looking at the IC column associated with each aspect, in all but one case, a change occurs. In 3.1 however, no change occurs in the IC, but instead the aspect is changed to flip the status of *BasePolicy* from interruptible to uninterruptible (as indicated by the commented line in Figure 2). The fact that this change is already captured for the IC is marked by the *AC* (*automatically captured*) in the IC column associated with the *Synchronization* aspect.

Looking at the rows associated with the restructurings, 5 change tasks apply to more than one aspect (1.4, 2.1, 2.2, 2.4, 3.3). In 3 of these, changes to ICs force changes in aspects (1.4, 2.1, 2.4), one of them is automatically and completely absorbed by pointcuts (2.2), and the other is automatically absorbed by pointcuts in two out of three aspects involved (3.3). Section 4.4 further considers changes that occur to more than one aspect. However, before considering those, the following subsections consider the positive/negative/neutral impact each of the aspects had on the evolution of the system as a whole. This section concludes with an overview of performance benchmarks on MMTk and MMTk_{ao} respectively.

4.1 Design Invariants: Assertions and Synchronization

In both cases, the aspects that encapsulate design invariants have a positive impact in that they provide a more precise and clearer representation of the internal structure of the crosscutting concerns. Each is in line with growth trends in the system. In the case of assertions, the (un)pluggable application of advice to all/no calls can be concisely and accurately represented. In the case of synchronization however, the pointcut enumerates a relatively long list, as the application of the advice must be selective instead of all/nothing.

In terms of negative impact, checking all assert/failure calls in MMTk_{ao} means there is some (small) amount of redundancy relative to MMTk, where several assertions can be made consecutively in a compound statement. With synchronization, the negative impact stems from the current lack of structure with respect to the otherwise exhaustive list.

Neutral impact, where MMTk and MMTk_{ao} tie in terms of evolvability, stems from the fact that changes to the location and existence of interacting concerns requires cosmetic updating of the objects (MMTk) and aspect (MMTk_{ao}) in a similar fashion. Even with the inversion of synchronization status in BasePolicy, there would have to be one change made, either to the aspect or the class. We consider this a tie, with the aspect having the slight edge because the nature of the change arguably falls within the realm of the crosscutting concern and not the interacting concern. Similarly, with respect to the renaming of the *VM_Uninterruptible* interface, in MMTk_{ao} this change was local to the aspect, whereas in MMTk system-wide search and replace would have to be applied throughout the code-base. The redistribution and elimination of the *VM_Interface* class during evolution also affected a design invariant, when the *Assert* class took over this functionality in the utility package, and one of the failure functions was eliminated. This change caused a renaming of all references to the function calls in the aspect in MMTk_{ao}, and throughout the code in MMTk.

Overall, however, we found that, though the *Synchronization* aspect does no worse than its scattered counterpart in terms of evolution, the *VerifyAssertions* aspect fares better due to its ability to grow/shrink correctly and precisely with the system.

4.2 GCSpy

Even though the aspect underwent numerous changes, its internal structure and external interaction on the whole was sustained throughout the evolution. In terms of positive impact, it was able to eliminate some redundant code relative to the subclassed *SemiSpace_With_GCSpy* in MMTk (section 2.1), and increase configurability by consolidating what were previously a collection of preprocessor directives coupled with global flags and subclassing.

In terms of negative impact, as a dynamic analysis tool, it is not surprising that GCSpy crosscuts multiple objects across multiple packages of the system. There is very little redundancy in the code captured by the GCSpy aspect, and thus there is a almost a 1:1 ratio of pointcut:advice definitions. Because GCSpy crosses structural and hierarchical boundaries in its interaction, it is subject to evolutionary changes at those interaction points. Among other things, GCSpy interacts with policies, multiple allocators, heap management, and the main collector thread. The evolution of the system caused changes to all of these interacting concerns as well as changes to previously non-interacting concerns. The addition of new policies to the system and the addition/removal of classes dealing with memory management forced changes in interaction.

The GCSpy concern was also affected by the relocation/renaming/redistribution of the VM_* classes, but the impact of this was no worse for the aspect than for the original code. Specifically the redistribution and elimination of the VM_Interface class in the system evolution required changes to all references made to its fields and methods in both MMTk and MMTk$_{ao}$. MMTk$_{ao}$ saw less change of this type due to the elimination of the VerifyAssertion and GCSpy field checks throughout the system. The case was the same for the relocation and renaming of the synchronization classes. This change required seven changes to the aspect across five advice and two inter-type declarations. Those same changes would have also taken place in the corresponding classes in MMTk. In this evolution the *VM_Address* class was also refactored, renamed, relocated changing parameters and return types of functions part of the GCSpy concern. These changes caused a refactoring of these functions and eliminated the use of the VM_Magic class. These changes again would be made in both the MMTk$_{ao}$ and MMTk.

Overall, the GCSpy aspect allows for improved evolution in MMTk$_{ao}$ due to the fact that it (1) sustained no more changes than the original implementation and (2) consolidates and unifies preprocessor directives/global flags/hierarchical decomposition as one manageable, locus of control for this dynamic analysis tool.

4.3 Prepare/Release Protocol

Though the positive impact of the prepare/release protocol aspect is the clarification of the design pattern, and this clarification holds throughout evolution, the negative impact involves the kinds of change that have to be made to the aspect as a result of evolving interacting concerns. The changes to the restructuring of the plan package to facilitate the move to unique naming of classes from the developers perspective had a negative impact on this aspect. Instead of being able to consolidate this combination of compiler directives and hierarchical decomposition, this aspect suffered from it. In

MMTk these changes are limited to the package itself and require no other changes in the system. MMTk$_{ao}$ leveraged the generic Plan.java naming convention in its original design. As a result of this change, the aspect for a given plan requires six occurrences of renaming change across four pointcuts and one advice. In the event that this prepare/release aspect is scaled across all plans, each of the plans would all have to have the corresponding renaming done to their pointcuts and advice.

4.4 Limitations and Future Work

Constructing MMTk$_{ao}$ allowed us to ask *what-if* with respect to a diverse set of aspects faced with big bang evolution. Future work will consider the costs of an initial refactoring for a system such as MMTk$_{ao}$, and the impact of tool support during the process of evolution. With respect to this case study, changes were performed by a single developer, and evolution of MMTk$_{ao}$ was dictated by the actual evolution of MMTk. It is reasonable to assume that given the original structure of MMTk$_{ao}$, evolution most likely would have played out differently than it did from MMTk. Additionally, the aspects themselves were uncovered by manual inspection of the control flow and the dominant decomposition of the system. Future work includes employing mining tools to identify further aspects.

This study provides a coarse-grained assessment of how a diverse set of aspects can be expected to fare during large-scale change to the system they crosscut. The refactoring in this case study was intentionally done in such a way as to be least invasive to the original system as possible. Based on our experience, we believe that to truly leverage the power of AOP in these examples, a more aggressive refactoring is required. For example, with respect to the semantics encompassed by the synchronization aspect, a stronger naming convention in the interacting concerns could influence the design of this aspect. Naming conventions are currently used within this system to impart design understanding to developers and could easily be used in this case to clarify which classes are in fact uninterruptible. This would facilitate the creation a more property based aspect that would provide a greater understanding of exactly what kind of classes fall into this synchronization family.

A further consideration for future work involves the fact that new concerns simultaneously impact multiple aspects. This confirms that the management issue of compositions of aspects requires a solution before the question of scalability can be more completely resolved.

Table 6 supplies a summary of the positive/negative/neutral effects of these aspects on evolution. The analysis presented here argues that the presence of aspects did not introduce any penalty in terms of the change tasks required, and that two out of four aspects provided evidence of better sustainability in that their structure facilitated evolution and further evolutionary trends.

4.5 Discussion

Although the four examples in this case study are limited, their diversity provides a basis for the categorization of aspect types and how each will hold up under system change. Specifically looking at the example of the dynamic analysis tool GCSpy, characterizations of the underlying nature of these types of tools – having a relatively large number of interaction points within a system that are necessarily scattered across many modules – begin to surface. It provides a general view of how an aspect with many interaction points will react when any or all these points are changed. Though future work includes a more detailed analysis of these and other kinds of characterizations, we begin some of that here.

Table 6. Impact of aspects on system evolution

ASPECT	POSITIVE	NEGATIVE	NEUTRAL
Verify Assertions (BETTER)	internal structure is clear plugability is useful all or nothing – all method calls captured by invariant	some redundant field checking relative to MMTk	change of method name used in a point-cut
Synchronization (no worse)	internal structure is clear and identifies a trend in design invari-ant, but only to a subset of classes (not all or nothing) localized change of and access to interface	must explicitly specify classes not to be cap-tured in aspect (no dominant pattern can be leveraged)	change in class-concern interaction automatically captures new classes
GCSpy (BETTER)	eliminates redundancy increased configurabil-ity	diverse interaction with interacting concerns	evolution of some interacting concerns
Prepare/Release (no worse)	highlights domain specific design pattern clarifies relationship of CCCs and its ICs with finite state machine	one aspect per plan evolution of MMTk yielded a similar result	evolution of interact-ing concerns

Looking at the results from the view given in Table 6, we can begin to generalize some of our findings and shed some light on what underlying characteristics might predispose certain kinds of aspects to positive/negative/neutral impact. The positive impact of each aspect in this study results from the accepted benefits associated with localization as applied to crosscutting concerns. The more debatable results are summarized in the final two columns detailing the negative/neutral effects.

In the design invariant aspects, the interaction points are numerous, but the behavior at those points is uniform and generalized. In this type of aspect, for example, Synchronization, the negative impact stems from the weak representation of the invariant, inviting scenarios that may lead to new code unintentionally be encompassed by the aspect. Again, with a more aggressive refactoring this abstraction may be improved and the problem may be alleviated. Additionally, with proper tool support the visibility of aspect interaction can be easily traced.

In the GCSpy and Prepare/Release aspects, the diversity of interaction fuels the negative impact in terms of changes that ripple from interacting concerns to aspects. The GCSpy aspect's diversity is the result of the varied responsibilities at each of its many interaction points, while the Prepare/Release aspect's diversity stems from the leveraging of the generic naming convention which encompasses all plan types. As a result of this diversity, negative/neutral impact encountered is tied to the number of changes that must be made to the aspect when these interacting concerns were renamed/relocated.

4.6 Fear of the Unknown: New Interactions, Multiple Aspects

No one can predict how a system will ultimately evolve. To get a slightly different perspective on the results of this study, we further categorized the changes into 3 groups: modification of interaction, elimination of interaction (where an interacting concern becomes a noninteracting concern, or *NIC*), and new interaction, as shown in the 3 columns in Table 7. Restructurings not listed in the table do not require changes to aspects.

Table 7 demonstrates that one aspect requires changes to existing interactions, two aspects eliminate interactions, and all four aspects deal with new interactions. Furthermore, of these new interactions, two out of four of them impact more than one aspect (2.1, 2.4).

Table 7. Changes by categories per aspect

IC_{OLD} ->IC_{NEW} CHANGE IN INTERACTION OF CCC WITH IC_{OLD}		**IC -> NIC** ELIMINATION OF INTERACTION OF CCC WITH IC		**NIC -> IC_{NEW}** INTRODUCTION OF INTERACTION OF CCC WITH PREVIOUSLY NIC	
CCC	Δ	**CCC**	Δ	**CCC**	Δ
GCSpy	1.1	GCSpy	1.2	Prepare/Release	2.1
	1.2		1.3	GCSpy	2.1
	1.4	VerifyAssertions	3.2		2.4
	2.1			VerifyAssertions	2.4
	3.3			Synchronization	1.4
					3.1

Arguably, these new interactions that require corresponding changes to multiple aspects potentially pose the biggest threat to developers leery of AOP. The fact that this data shows this kind of change impacts all the aspects in the study, coupled with the fact that these changes simultaneously impact multiple aspects, confirms that this indeed is an intensive, big bang style of evolution. Given that we believe this evolution scenario to be representative, if these kinds of changes cannot be effectively managed, the scalability of collections of aspects in this infrastructure is still an open question, and fertile ground for future work.

4.7 Performance

In the first set of tests considered here, the aspects included in MMTk$_{ao}$ are those that constitute its core functionality – *Prepare/Release* and *Synchronization.* Though the results show some noise (the average of three runs are reported), there is no discernable performance penalty for these aspects. The results of running tests from the DaCapo Benchmark Suite version beta050224 on the Jikes RVM v2.3.3, Linux 2.6.8-1.521smp, gcc-3.3.3-7, AspectJ v1.2, using an AMP Dual Athlon MP 2400+ machine with 1024 MB memory are shown below.

BENCHMARK	**MMTK**	**MMTk$_{ao}$**
Antlr	40708 ms	40937 ms (+0.56%)
Bloat	39752 ms	39550 ms (-0.51%)
Fop	14720 ms	14744 ms (0.16 %)
Jython	94148 ms	92297 ms (-1.97 %)
Pmd	42516 ms	43087 ms (1.34%)
Ps	87286 ms	86806 ms (-0.55%)

In order to stress-test a large, fine-grained aspect, we did a separate analysis of the *VerifyAssertions* aspect. In its current incarnation, these tests hammer the code within the aspect with 10s-100s of million invocations of assertion code, as reported below.

BENCHMARK	INVOCATIONS OF ASSERTION CODE
Antlr	74781288
Bloat	86211537
Fop	35325811
Jython	217938922
Pmd	99554310
Ps	100534321

Results from the fast path (assertions turned off) and the slow path (assertions turned on) introduces just under a 10% penalty[1]. In a future refactoring of the system

[1] An alternative implementation that could be effective at providing lighter weight support for this design invariant would be to rely on the *declare warning* construct in AspectJ to identify infractions at compile time, and at no cost.

we plan to move all assertion checking into a composition of aspects, so that when assertions are not required, the overhead would be removed.

Table 8. Fast path for assertions (off)

	ASSERTIONS OFF		
	MMTk (ms)	**MMTk$_{ao}$ (ms)**	**Increase (%)**
Antlr	37114	39824	7.3
Bloat	34346	37779	10
Fop	12794	14232	11.24
Jython	81689	88987	8.93
Pmd	38419	41460	7.92
Ps	78867	81020	2.73

Table 9. Slow path for assertions (on)

	ASSERTIONS ON		
	MMTk (ms)	**MMTk$_{ao}$ (ms)**	**Increase (%)**
Antlr	40708	46281	13.69
Bloat	39799	44183	11.02
Fop	14720	16787	14.04
Jython	93373	103508	10.85
Pmd	42516	47945	12.77
Ps	87286	91116	4.39

5 Conclusions

When evolution is a big bang, can aspects keep pace? This work provides a real world study comparing four crosscutting concerns in the original versus aspect-oriented implementation of the memory management subsystem within the Jikes RVM. One period of intense 10 month evolution of the dominant decomposition of the system is considered. This comparison highlights specific ways in which representative aspects have positive/negative/neutral impact on evolution. Given that aspects here did no harm in terms of a coarse-grained assessment of change tasks, and half of them did better than the original implementation, the study provides compelling evidence that aspects can indeed keep pace, and provide a means of better sustaining separation of concerns in system infrastructure software.

References

[1] E. Baniassad, G. Murphy, C. Schwanninger, and M. Kircher. Managing crosscutting concerns during software evolution tasks: an inquisitive study. In the Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), 2002.

[2] S. Blackburn, P. Chung and K. McKinley, Oil and Water? High Performance Garbage Collection in Java with MMTK, In the Proceedings of the International Conference on Software Engineering (ICSE), 2004.

[3] Y. Coady and G. Kiczales. A retroactive study of aspect evolution in operating system code. In the Proceedings of International Conference on Aspect-Oriented Software Development (AOSD), 2003.

[4] A. Colyer, A. Clement, Large-scale AOSD for Middleware. In the Proceedings of International Conference on Aspect-Oriented Software Development (AOSD), 2004.

[5] Dacapo Benchmarks, http://www-ali.cs.umass.edu/DaCapo/

[6] E. W. Dijkstra, A Discipline of Programming, Englewood Cliffs, United States: Prentice Hall, 1976.

[7] G. Duzan, J. Loyall, R. Schantz, Building Adaptive Distributed Applications with Middleware and Aspects. In the Proceedings of International Conference on Aspect-Oriented Software Development (AOSD) 2004.

[8] M. Engel, B. Freisleben, Supporting Autonomic Computing Functionality via Dynamic Operating System Kernel Aspects, In the Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), 2005.

[9] C. Gibbs and Y.Coady, *Aspect of Memory Management*, Hawaiin International Conference On System Sciences (HICSS), 2005.

[10] M.E. Fiuczynski, R. Grimm, Y.Coady, D. Walker, patch(1) Considered Harmful, The Tenth Annual Workshop on Hot Topics on Operating Systems (HotOS), 2005.

[11] B. Goetz, How does garbage collection work?, Developerworks, www-106.ibm.com/developerworks/java/library/j-jtp10283/, 2003.

[12] J. Hannemann and G. Kiczales. Design pattern implementations in Java and AspectJ. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), 2002.

[13] Haskell Compiler, http://www.haskell.org/ghc/

[14] IBM, AspectJ Project, http://eclipse.org/aspectj/, 2004.

[15] IBM, Jikes Research Virtual Machine, www-124.ibm.com/developerworks/oss/jikesrvm/, 2004.

[16] IBM, Jikes Research Virtual Machine User's Guide, www-124.ibm.com/developerworks/oss/jikesrvm/user-guide/HTML/userguide.html, 2004.

[17] M. Kersten and G. Murphy. Atlas: A case study. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), 1999.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier and J. Irwin, Aspect-Oriented Programming. In the Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP), 1997.

[19] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold, An overview of AspectJ. In the Proceedings of 15th European Conference on Object-Oriented Programming (ECOOP), 2001.

[20] L.L. Lehman and L.A. Belady, Program Evolution, APIC Studies in Data Processing, Volume 3, 1985.

[21] Gail C. Murphy, Lightweight Structural Summarization as an Aid to Software Evolution, Computer Science, University of Washington, PhD Thesis, 1996.

[22] G. Murphy, A. Lai, R. Walker, and M. Robillard. Separating features in source code: An Exploratory Study. In the Proceedins of the International Conference on Software Engineering (ICSE), 2001.
[23] OVM, http://www.ovmj.org/
[24] D.L. Parnas, On the Criteria To Be Used in Decomposing Systems into Modules, Communications of the ACM, 15(12), 1972.
[25] D.L. Parnas and P.C. Clements, Software State-of-the-Art: Selected Papers, in T. DeMarco and T. Lister, eds., A rational design process: How and why to fake it., Dorset House Publishing., 1990.
[26] A. Rashid, N.A. Leidenfrost: Supporting Flexible Object Database Evolution with Aspects. Generative Programming and Component Engineering (GPCE) 2004.
[27] W.P. Stevens, G.J. Meyers and L.L. Constantine, Structured Design, IBM Systems Journal, Volume 13, 1974.
[28] D. Sabbah, Aspects - from Promise to Realitys, Keynote, International Conference on Aspect-Oriented Software Development (AOSD), 2004.
[29] Sun, GCspy: A Generic Heap Visualisation Framework, research.sun.com/projects/GCSpy, 2004.
[30] A. Tesanovic, M. Amirijoo, M. Björk, J. Hansson, Empowering Configurable QoS Management in Real-Time Systems, International Conference on Aspect-Oriented Software Development (AOSD), 2005.
[31] R. Walker, E. Baniassad, and G. Murphy, An Initial Assessment of Aspect-Oriented Programming. In the Proceedings of the International Conference on Software. Engineering (ICSE), 1999.
[32] W. Wulf and Mary Shaw, Global variable considered harmful. SIGPLAN Notices, 8(2), 1973.
[33] C. Zhang, G. Gao, H.A. Jacobsen, Towards Just-in-time Middleware Architectures. In Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), 2005.

First-Class Relationships in an Object-Oriented Language

Gavin Bierman[1] and Alisdair Wren[2]

[1] Microsoft Research, Cambridge
`gmb@microsoft.com`
[2] University of Cambridge Computer Laboratory
`Alisdair.Wren@cl.cam.ac.uk`

Abstract. In this paper we investigate the addition of first-class relationships to a prototypical object-oriented programming language (a "middleweight" fragment of Java). We provide language-level constructs to declare relationships between classes and to manipulate relationship instances. We allow relationships to have attributes and provide a novel notion of relationship inheritance. We formalize our language giving both the type system and operational semantics and prove certain key safety properties.

1 Introduction

Object-oriented programming languages, and object modelling techniques more generally, provide software engineers with useful abstractions to create large software systems. The grouping of objects into classes and those classes into hierarchies provides the software engineer with an extremely flexible way of representing real-world semantic notions directly in code.

However, whilst object-oriented languages easily represent real-world entities (e.g. students, lectures, buildings), the programmer is poorly served when trying to represent the many natural *relationships* between those entities (e.g. 'attends lecture', 'is taught in').

Relationships clearly can be represented in object-oriented languages—indeed patterns have been established for the purpose [10]—but this important abstraction can get lost in the implementation that is forced upon the programmer by the lack of first-class support. Different aspects of the relationship can be implemented by fields and methods of the participating classes, but this distributes information about the relationship across various classes. Alternatively, small classes can be defined to contain references to the two related objects along with any attributes of the relationship. In both cases, without great care the structure can become internally inconsistent, especially in the presence of aliasing. Furthermore, we argue that the application of standard class-based inheritance to these 'relationship classes' does not adequately capture the intuitive semantics of relationship inheritance, which must otherwise be encoded in standard Java.

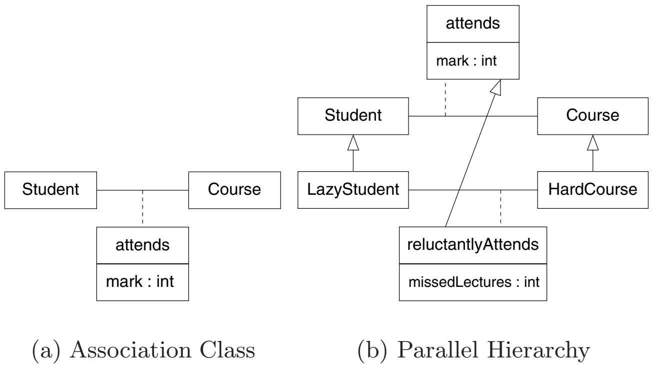


Fig. 1. Relationships represented as UML *association classes*

Such an encoding can only lead to further complexity and more opportunities for inconsistency.

The importance of relationships is clearly reflected by their prominence in almost all modelling languages: from (Extended) Entity-Relationship Diagrams (ER-diagrams) [5] to Unified Modelling Language (UML) [9]. In Figure 1 we give some examples of relationships expressed in UML (we use these as running examples throughout this paper).

We argue that such important abstractions deserve first-class support from programming languages. We are the not the first to do so; Rumbaugh also pointed out the importance of first-class language support for relationships [13]. Noble and Grundy also proposed that relationships should persist from the modelling to the implementation stage of program development [11]. Albano et al. propose a similar extension to a language for managing object-oriented databases (OODB) [1], but do so in a much richer data model and do not give a full description of their language.

In contrast to these works, our approach is more formal. We believe that such a formal, mathematical approach is essential to set a firm foundation for researchers, users and implementors of advanced programming languages. To that end, our main contribution is a precise description of how Java (or any other class-based, strongly-typed, object-oriented language) can be extended to support first-class relationships. Our tool is a small core language, RelJ, which is a subset of Java (much like Middleweight Java [4]) with suitable extensions for the support of relationships. RelJ provides means to define relationships between objects, to specify attributes associated with those relationships, and to create hierarchies of relationships. RelJ is intended to capture the essence of these extensions to Java, yet is small enough to formalize completely. Other features could be added to RelJ to make it a more complete language, but these would not impact on the extensions for relationships.

The remainder of the paper is organized as follows. In Sect. 2 we introduce our calculus and give a grammar. The type system of RelJ is defined in Sect. 3, where the formal notion of subtyping is discussed and well-typed RelJ programs are characterized. Section 4 gives the dynamics of RelJ with a small-step operational semantics. We outline a proof of type soundness for RelJ in Sect. 5. Section 6 describes an extension to RelJ which allows the addition of UML-style multiplicity restrictions to relationships. Finally, in Sect. 7, we conclude and consider further and related work.

2 The RelJ Calculus

As mentioned earlier, the core of RelJ is a subset of Java, similar to other fragments of Java-like languages [4, 7, 8]. The fragment we use consists of simple class declarations that contain a number of field declarations and method declarations. The exact form of the class declarations will be made more precise later.

2.1 Relationship Model

The main feature of RelJ is its support for first-class relationships. In addition to class declarations, therefore, a RelJ program consists of a number of relationship declarations, which are written:

$$\texttt{relationship}\ r\ \texttt{extends}\ r'\ \texttt{(}n\texttt{,}\ n'\texttt{)}\ \{\ \mathsf{FieldDecl}^*\ \mathsf{MethDecl}^*\ \}$$

This defines a relationship, r, with a number of type/field name pairs, $\mathsf{FieldDecl}^*$ and method declarations, $\mathsf{MethDecl}^*$. The relationship is between n and n' where n, n' range over classes *and* relationships. This provides a means for relationship instances to participate in further relationships. This feature is known as *aggregation* in ER-modelling [14]. An example is shown in Fig. 2: the `Recommends` relationship specifies that a `Tutor` may recommend a `Student` to attend a particular `Course` by relating an instance of `Tutor` to an instance of `Attends`, the relationship that specifies which students attend which courses. Relationships are directed (one-way) and many-to-many—more on this in Sect. 6.

We relate two objects, o_1 and o_2, with a relationship, r, by creating an instance of r, which then exists *between* o_1 and o_2, and stores the values for r's fields. Relationship instances are first-class runtime objects in RelJ and so can, for example, be stored in variables and fields. This immediately introduces design issues relating to the removal of relationship instances and consequent creation (or not) of dangling pointers: these are discussed later.

We also support relationship inheritance, which is denoted idiomatically in UML as inheritance between association classes (Fig. 1b). To the best of our knowledge, our support for this inheritance is novel and, as we will detail later, is significantly different from the standard class-based inheritance model.

2.2 Class Inheritance vs Relationship Inheritance

While class inheritance in RelJ is identical to that in Java, RelJ's relationship inheritance is based on a restricted form of delegation, as found in languages such as Self [16] and, more recently, δ [2]. Consider the RelJ code for a simple example, adapted from Pooley and Stevens [15], which is shown in Fig. 2.

```
class Student {
   String name;
}
class LazyStudent extends Student {
   int    hoursOfSleep;
}
class Course {
   String title;
}
class Tutor {
   String name;
}
relationship Attends (Student, Course) {
   int mark;
}
relationship ReluctantlyAttends extends Attends
                                (LazyStudent, Course) {
   int missedLectures;
}
relationship CompulsorilyAttends extends Attends
                                 (Student, Course) {
   String reason;
}
relationship Recommends (Tutor, Attends) {
   String reason;
}
...
alice = new LazyStudent();
programming = new Course();
typeSystems = new Course();
Attends.add(alice, programming);             // Alice attends Programming
ReluctantlyAttends.add(alice, typeSystems);
                                // Alice reluctantly attends Type Systems
for (Course c : alice.Attends) {
  print "Attends: " + c.title;
};                                        // Prints:
                                          //    Attends: Programming
                                          //    Attends: Type Systems
```

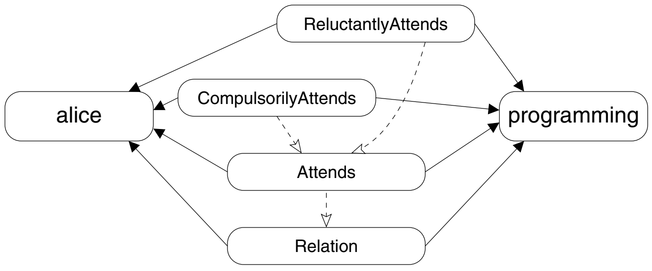


Fig. 2. Example RelJ code and possible instantiation

When `alice` and `programming` are placed in the `Attends` relationship, an instance of `Attends` is created between those objects. Subsequently, when `alice` and `programming` are further placed in `ReluctantlyAttends`, an instance of `ReluctantlyAttends` is created between `alice` and `programming`, but contains *only* the `missedLectures` field. If that `ReluctantlyAttends` instance receives a field look-up request for `mark`, it passes—*delegates*—the request to the `Attends` instance—the *super-instance*—that exists between those same objects.

To ensure all instances are 'complete', specifically that they have all the fields one would expect by inheritance, we impose the following invariant:

Invariant 1. *Consider a relationship r_2 which* ***extends*** *r_1. For every instance of relationship r_2 between objects o_1 and o_2, there is an instance of r_1, also between o_1 and o_2, to which it delegates requests for r_1's fields.*

By this invariant, if `alice` and `programming` were placed in the `ReluctantlyAttends` relationship without first having been placed in the `Attends` relationship, then an `Attends` instance would be implicitly created between them.

Invariant 2. *For every relationship r and pair of objects o_1 and o_2, there is at most one instance of r between o_1 and o_2.*

According to this second invariant, if `alice` and `programming` were later placed in the `CompulsorilyAttends` relationship, then its instance and that of `ReluctantlyAttends` would share a common super-instance: the `Attends` instance between `alice` and `programming`. This situation is shown at the bottom of Fig. 2, with the dotted lines indicating delegation of field lookups.

The motivation for such a mechanism is based on what one might intuitively expect from relationships: Clearly, if Alice reluctantly attends a course, then she also attends it and will receive a mark, thus we require sub-relationships to be included in their super-relationship, giving rise to Invariant 1. Also, if Alice is both compulsorily and reluctantly attending some course, the mark will be the same regardless of whether one views her attendance as reluctant, compulsory or without any annotation. Thus, for each pair of related objects, there should be only one instance of each relationship so that relationship properties are consistent, hence Invariant 2.

RelJ also allows the removal of relationship instances. For example, we could extend the code of Fig. 2 to remove the fact that `Alice` attends `programming`:

```
...
Attends.rem(alice, programming);  // Remove Alice attends Programming
for (Course c : alice.Attends){
  print "Attends: " + c.title;    // Prints:
}                                 //   Attends: Type Systems
```

In fact, both the relationship addition and removal operations are *statement expressions*. When used as an expression, `add` returns the relationship instance that was created: this provides a convenient short-cut for setting the new instance's fields. For regularity, `rem` returns the instance that was removed, or `null` if the relationship did not exist before the attempted removal.

We return now to the issue raised earlier concerning relationship instance removal. Consider the following code:

```
bob = new Student();
bob.name = "Bob";
databases = new Course();
databases.title = "DB 101";

bobdb = Attends.add(bob, databases);    // Add bob to databases
bobdb.mark = 99;
for (Course cs : bob.Attends) {
  print cs.title;
};                                      // Prints DB 101

print bobdb.mark;                       // Prints 99

Attends.rem(bob, databases);            // Remove bob from databases

for (Course cs : bob.Attends) {
  print cs.title;
};                                      // Prints nothing
```

The second iteration shows that the relationship between `bob` and `databases` has been correctly removed. We must then choose the fate of the reference to the `Attends`-instance stored in `bobdb`: what happens if we append the statement `print bobdb.mark;`?

There are clearly a number of options: either the instance is removed, in which case we would expect a runtime error; or the runtime maintains some liveness information so that an access to the variable `bobdb` would generate a specific relationship exception; or finally, we could choose not to remove the relationship instance at all, in which case the code would print `99`. We have chosen the third option. Thus, in RelJ, the relationship instance itself is not removed upon deletion, but rather is treated like any other runtime value and is removed by garbage collection. More experience in relationship programming is needed before we can determine if this is the correct design decision.

2.3 Language Definition

We give the grammar for RelJ programs and types in Fig. 3.

The Java types used in RelJ are class names and a single primitive type, `boolean` (the inclusion of further primitive types does not impact on the formalization). As discussed, we provide relationship names as types. To allow relationship processing RelJ has a (generic) set type `set<`n`>`, that denotes a set of values of type n. This set type is not a *reference* type, but is a *primitive*

$p \in$ Program	$::=$	ClassDecl* RelDecl*	
ClassDecl	$::=$	`class` c `extends` c' { FieldDecl* MethDecl* }	
RelDecl	$::=$	`relationship` r `extends` r' `(`n`,` n'`)` { FieldDecl* MethDecl* }	
$n \in$ NominalType	$::=$	$c \mid r$	
$t \in$ Type	$::=$	`boolean` $\mid n \mid$ `set<`n`>`	
FieldDecl	$::=$	$t\ f$`;`	
MethDecl	$::=$	$t\ m$`(`$t'\ x$`)` mb	
$mb \in$ MethBody	$::=$	`{` s `return` e`;` `}`	
$v \in$ Value	$::=$	`true` $\mid$ `false` $\mid$ `null` $\mid$ `empty`	
$l \in$ LValue	$::=$	$x \mid$	
		e`.`f	field access
$e \in$ Expression	$::=$	$v \mid$	value
		$l \mid$	l-value
		e_1 `==` $e_2 \mid$	equality test
		e_1 `+` $e_2 \mid e_1$ `-` $e_2 \mid$	set addition/removal
		e`.`$r \mid e$`:`$r \mid$	relationship access
		e`.from` $\mid$	relationship source
		e`.to` $\mid$	relationship destination
		se	statement expression
$se \in$ StatementExp	$::=$	`new` c`()` $\mid$	instantiation
		l `=` $e \mid$	assignment
		r`.add(`e`,`e'`)` $\mid r$`.rem(`e`,`e'`)` $\mid$	relationship addition/removal
		e`.`m`(`e'`)`	method call
$s \in$ Statement	$::=$	$\epsilon \mid$	empty statement
		se`;` $s_1 \mid$	expression
		`if (`e`)` $\{s_1\}$ `else` $\{s_2\}$`;` $s_3 \mid$	conditional
		`for (`$n\ x$ `:` e`)` $\{s_1\}$`;` s_2	set iteration

Fig. 3. The grammar of RelJ types and programs

(value) type, much like the generic literal types used by the ODMG [12].[1] RelJ does not support nested sets—sets of sets are not permitted. RelJ offers a `for` iterator over set values (we adopt the same syntax as Java 5.0 for iterating over collections). We also provide operators for explicitly adding an element to a set (`+`), and for removing an element (`-`).

[1] Having sets as a generic value type allows us to soundly support covariance—this is discussed in more detail in Sect. 3.

$$\begin{aligned}
\mathcal{C} \in \mathsf{ClassTable} &: \mathsf{ClassName} \rightarrow \mathsf{ClassName} \times \mathsf{FieldMap} \times \mathsf{MethMap}\\
\mathcal{R} \in \mathsf{RelTable} &: \mathsf{RelName} \rightarrow \mathsf{RelName} \times \mathsf{NominalType} \times \mathsf{NominalType} \times\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{FieldMap} \times \mathsf{MethMap}\\
\mathcal{F} \in \mathsf{FieldMap} &: \mathsf{FldName} \rightarrow \mathsf{Type}\\
\mathcal{M} \in \mathsf{MethMap} &: \mathsf{MethName} \rightarrow \mathsf{VarName} \times \mathsf{LocalMap} \times \mathsf{Type} \times \mathsf{Type} \times \mathsf{MethBody}\\
\mathcal{L} \in \mathsf{LocalMap} &: \mathsf{VarName} \rightarrow \mathsf{Type}
\end{aligned}$$

Fig. 4. Signatures of class and relationship tables

For simplicity, we require some regularity in the class (and relationship) declarations of RelJ programs: (1) we insist that all class declarations include the supertype; (2) we write out the receiver of field access or method invocation in full; (3) all methods take just one argument; (4) all method declarations end with a `return` statement; and (5) we assume that in a RelJ program exactly one class supports a `main` method. To be concise, we do not consider constructor methods; field initialization, other than the provision of type-appropriate initial values, is performed explicitly.

The metavariable c ranges over the set of class names, ClassName; r ranges over the set of relationship names, RelName; n ranges over both ClassName and RelName; f ranges over the set of field names, FldName; m ranges over the set of method names, MethName; and x ranges over the set of variable names, VarName, which we assume contains the element `this`, which cannot be on the left-hand side of an assignment. Metavariables may not take the undefined value.

As usual for such language formalizations, we assume that given a RelJ program, P, the class and relationship declarations give rise to class and relationship tables that are denoted by $\mathcal{C}_P$ and $\mathcal{R}_P$, respectively [6]. (We will drop the subscript when it is unambiguous.) A class (relationship) table is then a map from a class (relationship) name to a class (relationship) definition. Signatures for these maps are to be found in Fig. 4.

A class definition is a tuple, $(c, \mathcal{F}, \mathcal{M})$, where c is the superclass; $\mathcal{F}$ is a map from field names to field types; and $\mathcal{M}$ is a map from method names to method definitions. Method definitions are tuples $(x, \mathcal{L}, t_1, t_2, mb)$ where x is the parameter; $\mathcal{L}$ is a map from local variable names to their types; t_1 is the parameter type; t_2 is the return type; and mb is the method body. For brevity, we write $\mathcal{F}_c$ and $\mathcal{M}_c$ for the field and method definition maps of class c.

Relationship definitions are tuples $(r', n, n', \mathcal{F}, \mathcal{M})$ where r' is the super-relationship; n and n' are the types between which the relationship is formed (the *source* and *destination* respectively); and $\mathcal{F}$, $\mathcal{M}$ are the field map and method map respectively, as found in class definitions. As for classes, we write $\mathcal{F}_r$ for r's field definition map and $\mathcal{M}_r$ for r's method map.

In summary, RelJ offers the following operations to manipulate relationships: e`.`r finds the objects related to the result of e through relationship r; e`:`r finds the instances of r that exist between the result of e and the objects to which it is related; and the pseudo-fields `from` and `to` are made available on relationship instances, and return the source and destination objects between which the instance exists (or existed). These are further described in the following sections.

3 Type System

We provide `Object` for the root of the class hierarchy as usual, and `Relation` as its counterpart in the relationship hierarchy, and assume appropriate entries in $\mathcal{C}$ and $\mathcal{R}$ respectively. We define the usual subtyping relation $P \vdash t \leq t'$ where t is a subtype of t', directly populated with the information about immediate super-types provided by $\mathcal{C}$ and $\mathcal{R}$, then closed under transitivity and reflexivity. P is omitted where the context makes it unambiguous.

We leave the less important typing rules to Appendix A, but two rules worth particular note are shown here:

$$\frac{\vdash n_1 \leq n_2}{\vdash \texttt{set<}n_1\texttt{>} \leq \texttt{set<}n_2\texttt{>}}\ (\text{STCov}) \qquad \frac{}{\vdash \texttt{Relation} \leq \texttt{Object}}\ (\text{STObject})$$

STCov makes set types covariant with their contained type. If $\texttt{set<}-\texttt{>}$ were a reference type, then this kind of covariance would be unsound. However, $\texttt{set<}-\texttt{>}$ is a value type, thus such values are not referenced or mutated, only copied.

To unify the relationship and class hierarchies—desirable in the absence of generics—we take `Relation` as a subtype of `Object` in rule STObject.[2]

While $\mathcal{F}_c$ and $\mathcal{M}_c$ give us the fields and methods declared directly in c, we define $\mathcal{FD}_c$ and $\mathcal{MD}_c$ to provide us with all the fields and methods available for c's instances, including those inherited from its superclasses, so that their types might be checked in the later type rules:

$$\mathcal{FD}_c(f) = \begin{cases} \mathcal{F}_c(f) & \text{if } f \in \mathsf{dom}(\mathcal{F}_{P,c}) \text{ or } c = \texttt{Object} \\ \mathcal{FD}_{c'}(f) & \text{if } f \notin \mathsf{dom}(\mathcal{F}_{P,c}) \text{ and } \mathcal{C}(c) = (c', \_, \_) \end{cases}$$

$\mathcal{MD}$ is defined similarly for class methods, as are $\mathcal{FD}$ and $\mathcal{MD}$ for relationships.

We type expressions and statements in the presence of a typing environment, Γ, which assigns types to variable names. Selected typing judgements for RelJ expressions are given below:

$$\frac{\Gamma \vdash e\colon n_1 \quad \mathcal{R}(r) = (\_, n_2, n_3, \_, \_) \quad \vdash n_1 \leq n_2}{\Gamma \vdash e\texttt{.}r\colon \texttt{set<}n_3\texttt{>}}\ (\text{TSRelObj}) \qquad \frac{\Gamma \vdash e\colon n_1 \quad \mathcal{R}(r) = (\_, n_2, \_, \_, \_) \quad \vdash n_1 \leq n_2}{\Gamma \vdash e\texttt{:}r\colon \texttt{set<}r\texttt{>}}\ (\text{TSRelInst})$$

TSRelObj types the lookup of objects related through r to the result of e. As our relationships are implicitly many-to-many, the result of this lookup is a set of r's destination type, n_2. The relationship instances that sit between the result of e and the result of $e\texttt{.}r$ are accessed through $e\texttt{:}r$. The result of such a lookup is a set of r-instances, as specified in TSRelInst. There is a bias here between the source and destination of a relationship: the relationship instances may only be accessed from the source object. It is not difficult to extend the language so that access from the destination objects is also possible.

[2] If we added generics to RelJ it would be possible to remove this typing rule.

$$\frac{\begin{array}{c}\text{(TSFrom)}\\ \Gamma \vdash e\colon r\\ \mathcal{R}(r) = (\_, n, \_, \_, \_)\end{array}}{\Gamma \vdash e\texttt{.from}\colon n} \qquad \frac{\begin{array}{c}\text{(TSTo)}\\ \Gamma \vdash e\colon r\\ \mathcal{R}(r) = (\_, \_, n, \_, \_)\end{array}}{\Gamma \vdash e\texttt{.to}\colon n}$$

Given an r-instance, the objects between which it exists (or between which it once existed) can be accessed with the `from` and `to` properties. TSFrom and TSTo assign types according to the relationship's declaration—therefore, these are typed covariantly with the relationship type, but this is sound as they are immutable for all instances of such a relationship.

$$\frac{\begin{array}{c}\text{(TSRelAdd)}\\ \mathcal{R}(r) = (\_, n_1, n_2, \_, \_)\\ \Gamma \vdash e_1 : n_3\\ \Gamma \vdash e_2 : n_4\\ \vdash n_3 \leq n_1\\ \vdash n_4 \leq n_2\end{array}}{\Gamma \vdash r\texttt{.add(}e_1\texttt{,}e_2\texttt{)} : r} \qquad \frac{\begin{array}{c}\text{(TSRelRem)}\\ \mathcal{R}(r) = (\_, n_1, n_2, \_, \_)\\ \Gamma \vdash e_1 : n_3\\ \Gamma \vdash e_2 : n_4\\ \vdash n_3 \leq n_1\\ \vdash n_4 \leq n_2\end{array}}{\Gamma \vdash r\texttt{.rem(}e_1\texttt{,}e_2\texttt{)} : r}$$

Finally, TSRelAdd and TSRelRem specify typing of the operators that relate and unrelate objects. In both cases, e_1 and e_2 must be of the source and destination type, respectively, of relationship r. The result of either operation will be an instance of r; that which was created or removed. A removal may evaluate to `null` where the results of e_1 and e_2 were unrelated by r.

The type-checking relation for statements is of the form $\Gamma \vdash s$, the rules for which are largely routine. We show some examples, however:

$$\frac{\begin{array}{c}\text{(TSExp)}\\ \Gamma \vdash se\colon t\\ \Gamma \vdash s\end{array}}{\Gamma \vdash se\texttt{;}\ s} \qquad \frac{\begin{array}{c}\text{(TSFor)}\\ \Gamma \vdash e\colon \texttt{set<}n_1\texttt{>}\\ \Gamma[x \mapsto n_2] \vdash s_1\\ \vdash n_1 \leq n_2\\ \Gamma \vdash s_2\end{array}}{\Gamma \vdash \texttt{for (}n_2\ x\ \texttt{:}\ e\texttt{)}\ \{s_1\}\texttt{;}\ s_2}\, x \notin \mathsf{dom}(\Gamma)$$

TSExp allows type-correct statement expressions to be used as statements, while TSFor checks that the `for` construct is only asked to iterate over a set of object references. Note that, to be consistent with the Java 5.0 syntax, we require an explicit type for the iterating variable, although there is no reason why this type could not be inferred. We also require that the iteration variable is not already in scope.

The set $\mathsf{validTypes}_P$ specifies the types that may be assigned to fields and variables:

$$\mathsf{validTypes}_P = \{\texttt{boolean}\} \cup \mathsf{dom}(\mathcal{C}_P) \cup \mathsf{dom}(\mathcal{R}_P) \cup \{\texttt{set<}n\texttt{>} \mid n \in \mathsf{dom}(\mathcal{C}_P) \cup \mathsf{dom}(\mathcal{R}_P)\}$$

In the following two rules, we check fields and methods in the presence of their enclosing class or relationship:

$$\frac{\begin{array}{ll} & \mathcal{C}(n) = (n', \_, \_) \vee \mathcal{R}(n) = (n', \_, \_, \_, \_) \\ 1. & f \notin \mathsf{dom}(\mathcal{FD}_{n'}) \\ 2. & \mathcal{F}_n(f) \in \mathsf{validTypes}_P \\ 3. & \mathcal{R}(f) = (\_, n_1, n_2, \_) \Rightarrow \not\vdash n \leq n_1 \end{array}}{P, n \vdash f} \quad \text{(TSField)}$$

TSField checks that f is a good field for class or relationship n by verifying (1) that f is not defined in any super-type of n; (2) that f's type is valid in a well-typed program and (3) that there is no relationship with the same name as f that might make references to f ambiguous.

$$\frac{\begin{array}{ll} & \mathcal{C}_P(n) = (n', \_, \mathcal{M}_n) \vee \mathcal{R}_P(n) = (n', \_, \_, \_, \mathcal{M}_n) \\ & \mathcal{M}_n(m) = (x, \mathcal{L}, t_1, t_2, \{\ s\ \texttt{return}\ e;\ \}) \\ 1. & t_1 \in \mathsf{validTypes}_P \\ 2. & \texttt{this}, x \notin \mathsf{dom}(\mathcal{L}) \\ 3. & \{x \mapsto t_1, \texttt{this} \mapsto n\} \cup \mathcal{L} \vdash s \\ 4. & \{x \mapsto t_1, \texttt{this} \mapsto n\} \cup \mathcal{L} \vdash e : t_2' \\ 5. & \vdash t_2' \leq t_2 \\ 6. & \mathcal{MD}_{n'}(m) = (\_, \_, t_3, t_4, \_) \Rightarrow\ \vdash t_3 \leq t_1\ \wedge\ \vdash t_2 \leq t_4 \end{array}}{P, n \vdash m} \quad \text{(TSMethod)}$$

TSMethod checks (1) that the input type of method m in class/relationship n is valid; (2) that the parameter name and `this` do not clash with any local variables; (3) that the method body is well-typed when the parameter, `this` and the local variables are assigned the types specified in the class' method table; (4, 5) that the `return` expression has a subtype of the method's declared return type; and (6) that the input type of this method is a supertype of any previous declaration of m in a super-type of c, and that the return type of m is a subtype of any previous method declaration: that is, that this definition of m may be used anywhere a supertype's version of m can be used. We then specify the validity of classes and relationships:

$$\frac{\begin{array}{c} \mathcal{C}(c) = (c' \neq c, \mathcal{F}, \mathcal{M}) \\ P \vdash c' \\ \forall f \in \mathsf{dom}(\mathcal{F}) : P, c \vdash f \\ \forall m \in \mathsf{dom}(\mathcal{M}) : P, c \vdash m \end{array}}{P \vdash c} \quad \text{(TSClass)}$$

$$\frac{\begin{array}{ll} & \mathcal{R}_P(r) = (r' \neq r, n_1, n_2, \mathcal{F}, \mathcal{M}) \\ & r' \in \mathsf{validTypes}_P \\ 1. & \mathcal{R}_P(r') = (\_, n_1', n_2', \_, \_) \\ 2. & \vdash n_1 \leq n_1' \\ 3. & \vdash n_2 \leq n_2' \\ & \forall f \in \mathsf{dom}(\mathcal{F}) : P, r \vdash f \\ & \forall m \in \mathsf{dom}(\mathcal{M}) : P, r \vdash m \end{array}}{P \vdash r} \quad \text{(TSRelationship)}$$

TSClass specifies that a class type is well-formed if its superclass is well-formed, and if all of its methods and fields are well-typed. TSRelationship imposes many of the same restrictions as TSClass, with the addition of conditions 1–3, which check the types related by r's super-relationship are supertypes of those that r relates.

4 Semantics

We specify evaluation rules for a small-step semantics. We use evaluation contexts to specify evaluation order [17], and use variable renaming to avoid the need for an explicit frame stack [7].

The meta-variables used in the semantics range over addresses, values, errors, objects and stores as follows:

$$\begin{aligned}
\iota &\in \mathsf{Address}\\
\iota^{\texttt{null}} &\in \mathsf{Address} \cup \{\texttt{null}\}\\
u &\in \mathsf{DynValue} = \{\texttt{null}, \texttt{true}, \texttt{false}\} \cup \mathsf{Address} \cup \mathcal{P}(\mathsf{Address})\\
w &\in \mathsf{Error} ::= \mathsf{NullPtrError} \mid \mathcal{E}_{\mathrm{e}}[w] \mid \mathcal{E}_{\mathrm{s}}[w] \mid \{\ w\ \texttt{return}\ e\texttt{;}\ \}\\
o &\in \mathsf{Object}\\
\sigma &: \mathsf{Address} \rightarrow \mathsf{Object}\\
\rho &: (\mathsf{Address} \times \mathsf{Address} \times \mathsf{RelName}) \rightarrow \mathsf{Address}\\
\lambda &: \mathsf{VarName} \rightarrow \mathsf{DynValue}
\end{aligned}$$

Objects, ranged over by o, are either class instances or relationship instances. We write class instances as an annotated pair, $\langle\!\langle c \| f_1 : v_1, \ldots, f_i : v_i \rangle\!\rangle$, containing a mapping from field names to values, and the object's dynamic type, c. Relationship instances are written as an annotated 5-tuple, $\langle\!\langle r, \iota^{\texttt{null}}, \iota_1, \iota_2 \| f_1 : v_1, \ldots, f_i : v_i \rangle\!\rangle$, containing the familiar field value map and dynamic type, as well as the object addresses the instance relates, ι_1 and ι_2, and a reference to the relationship instance's *super-instance*, $\iota^{\texttt{null}}$; specifically, the instance of r's super-relationship which relates the same object addresses ι_1 and ι_2. Where $r = \texttt{Relation}$, there is no super-relationship and this reference is `null`. For both types of object, we take $o(f)$ and $\mathsf{dom}(o)$ as if they were applied to o's field value map.

Dynamic values (as opposed to syntactic value literals), ranged over by u, are either addresses, ranged over by ι, sets of addresses, or `true`, `false` or `null`. A small-step semantics means that expressions may at times be only partially evaluated, so we include these run-time values and partially-evaluated method bodies in language expressions by extending Expression as follows:

$e \in \mathsf{DynExpression} ::=$	
$u \mid$	dynamic values
$mb \mid$	method body
$\ldots$	terms from Expression grammar

DynLValue and DynStatement are generated from LValue and Statement in the obvious way, and e, l and s will range over these new definitions from this point onward.

A store, σ, is a map from addresses to objects, while local variables are given values by a locals store, λ. A relationship store, ρ maps relationship tuples to addresses such that $\rho(r, \iota_1, \iota_2)$ indicates the address of the instance of r which exists between ι_1 and ι_2.

During execution, the store and its constituent objects are modified by updating the relevant map. Update of some map f is written $f[a \mapsto b]$ such that

$\mathcal{E}_\mathrm{e} \in \mathsf{ExpContext} ::=$	
$\bullet$	hole
$\mid \mathcal{E}_\mathrm{e}\texttt{.}f$	field lookup
$\mid \mathcal{E}_\mathrm{e}\ \texttt{==}\ e \mid u\ \texttt{==}\ \mathcal{E}_\mathrm{e}$	equality test
$\mid \mathcal{E}_\mathrm{e}\ \texttt{+}\ e \mid u\ \texttt{+}\ \mathcal{E}_\mathrm{e}$	set addition
$\mid \mathcal{E}_\mathrm{e}\ \texttt{-}\ e \mid u\ \texttt{-}\ \mathcal{E}_\mathrm{e}$	set removal
$\mid \mathcal{E}_\mathrm{e}\texttt{.}r \mid \mathcal{E}_\mathrm{e}\texttt{:}r$	relationship access
$\mid \mathcal{E}_\mathrm{e}\texttt{.from} \mid \mathcal{E}_\mathrm{e}\texttt{.to}$	relationship from/to
$\mid \texttt{\{}\ \mathcal{E}\ \texttt{return}\ e\texttt{;}\ \texttt{\}} \mid \texttt{\{}\ \texttt{return}\ \mathcal{E}_\mathrm{e}\texttt{;}\ \texttt{\}}$	method body
$\mid \mathcal{E}_\mathrm{e}\texttt{.}f\ \texttt{=}\ e \mid x\ \texttt{=}\ \mathcal{E}_\mathrm{e} \mid u\texttt{.}f\ \texttt{=}\ \mathcal{E}_\mathrm{e}$	assignment
$\mid \mathcal{E}_\mathrm{e}\texttt{.}m\texttt{(}e'\texttt{)} \mid u\texttt{.}m\texttt{(}\mathcal{E}_\mathrm{e}\texttt{)}$	method call
$\mid r\texttt{.add(}\mathcal{E}_\mathrm{e}\texttt{,}e'\texttt{)} \mid r\texttt{.add(}u\texttt{,}\mathcal{E}_\mathrm{e}\texttt{)}$	relationship addition
$\mid r\texttt{.rem(}\mathcal{E}_\mathrm{e}\texttt{,}e'\texttt{)} \mid r\texttt{.rem(}u\texttt{,}\mathcal{E}_\mathrm{e}\texttt{)}$	relationship removal
$\mathcal{E}_\mathrm{s} \in \mathsf{StatContext} ::=$	
$\mathcal{E}_\mathrm{e}\texttt{;}\ s$	expression
$\mid \texttt{for (}n\ x\ \texttt{:}\ \mathcal{E}_\mathrm{e}\texttt{)}\ \{s_1\}\texttt{;}\ s_2$	set iteration
$\mid \texttt{if (}\mathcal{E}_\mathrm{e}\texttt{)}\ \{s_1\}\ \texttt{else}\ \{s_2\}\texttt{;}\ s_3$	conditional

Fig. 5. Grammar for evaluation contexts

$f[a \mapsto b](a) = b$ and $f[a \mapsto b](c) = f(a)$ where $a \neq c$. Such substitutions are commonly applied to stores ($\sigma[\iota \mapsto o]$) and to objects ($o[f \mapsto v]$).

Substitution of variables in program syntax uses the standard notation, $e[x'/x]$, for the replacement of all variables x in e with x', and similarly with statements, $s[x'/x]$.

Figure 5 gives the evaluation contexts for RelJ expressions and statements. All contexts $\mathcal{E}$ contain a hole, denoted $\bullet$, which indicates the position of the sub-expression to be evaluated first—in this case the left-most, inner-most. An expression may be placed in a context's hole position by substitution, denoted $\mathcal{E}_\mathrm{e}[e]$. Notice that we no longer distinguish between those expressions that may or may not be used in statement position.

A *configuration* in the semantics is a 5-tuple of typing environment, heap, relationship store, locals map, and a statement: $\langle \Gamma, \sigma, \rho, \lambda, s\rangle$. An *error configuration* is a configuration $\langle \Gamma, \sigma, \rho, \lambda, w\rangle$, with an error in place of a statement. Γ is included for the proof of type soundness.

Expression execution proceeds when a sub-expression in hole position may be reduced, as specified by OSContextE. We elide the similar rule for expressions in statement context:

$$(\textsc{OSContextE})\ \frac{\langle \Gamma, \sigma, \rho, \lambda, e\rangle \overset{P}{\rightsquigarrow} \langle \Gamma', \sigma', \rho', \lambda', e'\rangle}{\langle \Gamma, \sigma, \rho, \lambda, \mathcal{E}_e[e]\rangle \overset{P}{\rightsquigarrow} \langle \Gamma', \sigma', \rho', \lambda', \mathcal{E}_e[e']\rangle}$$

We also execute statements inside partially-executed method bodies:

$$(\textsc{OSInBody})\ \frac{\langle \Gamma, \sigma, \rho, \lambda, s\rangle \overset{P}{\rightsquigarrow} \langle \Gamma', \sigma', \rho', \lambda', s'\rangle}{\langle \Gamma, \sigma, \rho, \lambda, \texttt{\{}\ s\ \texttt{return}\ e\texttt{;}\ \texttt{\}}\rangle \overset{P}{\rightsquigarrow} \langle \Gamma', \sigma', \rho', \lambda', \texttt{\{}\ s'\ \texttt{return}\ e\texttt{;}\ \texttt{\}}\rangle}$$

It remains now to define the base cases for the operational semantics. We begin with RelJ's two relationship operations on an object address, ι: firstly, the

$$\mathsf{newPart}_P(r, \iota^{\mathtt{null}}, \iota_1, \iota_2) = \langle\!\langle r, \iota^{\mathtt{null}}, \iota_1, \iota_2 \| f_1 : \mathsf{initial}_P(\mathcal{F}_{P,r}(f_1)), \ldots, f_i : \mathsf{initial}_P(\mathcal{F}_{P,r}(f_i))\rangle\!\rangle$$
$$\text{where } \{f_1, f_2, \ldots, f_i\} = \mathsf{dom}(\mathcal{F}_{P,r})$$

$$\mathsf{addRel}_P(r, \iota_1, \iota_2, \sigma_1, \rho_1) = \begin{cases} (\sigma_1, \rho_1) & \text{if } \rho(r, \iota_1, \iota_2) = \iota'' \\ (\sigma_1[\iota \mapsto \mathsf{newPart}_P(r, \mathtt{null}, \iota_1, \iota_2)], \rho_1[(r, \iota_1, \iota_2) \mapsto \iota]) & \\ & \text{if } r = \mathtt{Relation} \\ (\sigma_3, \rho_3) & \text{otherwise} \end{cases}$$
$$\begin{aligned} &\text{where } \iota \notin \mathsf{dom}(\sigma_1) \text{ or } \mathsf{dom}(\sigma_2) \\ &\quad r \neq \mathtt{Relation} \Rightarrow \mathcal{R}_P(r) = (r', \_, \_, \_) \\ &\quad (\sigma_2, \rho_2) = \mathsf{addRel}_P(r', \iota_1, \iota_2, \sigma_1, \rho_1) \\ &\quad \sigma_3 = \sigma_2[\iota \mapsto \mathsf{newPart}_P(r, \rho_2(r', \iota_1, \iota_2), \iota_1, \iota_2)] \\ &\quad \rho_3 = \rho_2[(r, \iota_1, \iota_2) \mapsto \iota] \end{aligned}$$

$$\mathsf{remRel}_P(r, \iota_1, \iota_2, \rho) = \rho \setminus \{((r', \iota_1, \iota_2) \mapsto \iota) \mid \vdash r' \leq r\}$$

$$\mathsf{fldUpd}(\sigma, f, \iota, u) = \begin{cases} \sigma[\iota \mapsto \sigma(\iota)[f \mapsto u]] & \text{if } f \in \mathsf{dom}(\sigma(\iota)) \\ \mathsf{fldUpd}(\sigma, f, \iota', u) & \text{if } \sigma(\iota) = \langle\!\langle r, \iota', \_, \_ \| \ldots \rangle\!\rangle \end{cases}$$

Fig. 6. Definitions of auxiliary functions for creating relationship instances (newPart), for putting objects in relationships (addRel) and for removing objects from relationships (remRel). fldUpd demonstrates delegation of field updates to super-relationship instances

objects related to ι by relationship r may be accessed using $e\texttt{.}r$; secondly, the instances of r that relate those objects to ι may be accessed with $e\texttt{:}r$ so that relationship attributes may be read or modified:

OSRelObj: $\langle \Gamma, \sigma, \rho, \lambda, \iota\texttt{.}r\rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \{\iota' \mid \exists \iota'' : \rho(r, \iota, \iota') = \iota''\}\rangle$
OSRelObjN: $\langle \Gamma, \sigma, \rho, \lambda, \mathtt{null}\texttt{.}r\rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError}\rangle$
OSRelInst: $\langle \Gamma, \sigma, \rho, \lambda, \iota\texttt{:}r\rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \{\iota'' \mid \exists \iota' : \rho(r, \iota, \iota') = \iota''\}\rangle$

OSRelObj and OSRelObjN give the semantics for obtaining the objects related to ι through r. Notice that the result is not just a matter of looking-up the result in a table; the objects are found by querying ρ. If `null` is the target of the lookup, a null-pointer error occurs. Similar rules are left for the appendix.

The pseudo-fields `from` and `to` provide access to the objects between which a relationship instance exists, returning the source and destination objects respectively:

OSFrom: $\langle \Gamma, \sigma, \rho, \lambda, \iota\texttt{.from}\rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \iota'\rangle$ where $\sigma(\iota) = \langle\!\langle \_, \_, \iota', \_ \| \_\rangle\!\rangle$
OSTo: $\langle \Gamma, \sigma, \rho, \lambda, \iota\texttt{.to}\rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \iota'\rangle$ where $\sigma(\iota) = \langle\!\langle \_, \_, \_, \iota' \| \_\rangle\!\rangle$

OSRelAdd and OSRelRem give semantics to the relationship addition and removal operators `add` and `rem` respectively, and are based entirely on addRel and remRel from Fig. 6:

OSRelAdd: $\langle \Gamma, \sigma_1, \rho_1, \lambda, r\texttt{.add(}\iota_1\texttt{,}\iota_2\texttt{)}\rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma_2, \rho_2, \lambda, \iota_3\rangle$
where $(\sigma_2, \rho_2) = \mathsf{addRel}_P(r, \iota_1, \iota_2, \sigma_1, \rho_1)$ and $\iota_3 = \rho_2(r, \iota_1, \iota_2)$

OSRelRem1: $\langle \Gamma, \sigma, \rho_1, \lambda, r\texttt{.rem(}\iota_1\texttt{,}\iota_2\texttt{)}\rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho_2, \lambda, \rho_1(r, \iota_1, \iota_2)\rangle$
where $(r, \iota_1, \iota_2) \in \mathsf{dom}(\rho_1)$ and $\rho_2 = \mathsf{remRel}_P(r, \iota_1, \iota_2, \rho_1)$

OSRelRem2: $\langle \Gamma, \sigma, \rho, \lambda, r\texttt{.rem(}\iota_1\texttt{,}\iota_2\texttt{)}\rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \texttt{null}\rangle$
where $(r, \iota_1, \iota_2) \notin \mathsf{dom}(\rho)$

addRel adds an instance of r between ι_1 and ι_2 if such an instance does not already exist. With a recursive call, it also ensures that instances of r's super-relationships exist between ι_1 and ι_2, ensuring Invariant 1 is maintained.

remRel removes an instance of r from between ι_1 and ι_2, but does *not* alter the heap, only the relationship store, ρ. Again, to maintain Invariant 1, all instances of sub-relationships to r are similarly removed from between ι_1 and ι_2.

In the case of a relationship addition in expression context, a reference is returned to the relationship instance that was added. Relationship removal evaluates to the instance that was removed, if any. Where no such instance exists, `null` is returned.

Field update is performed with an auxiliary function fldUpd, also found in Fig. 6, which demonstrates the delegation of field lookup to super-relationship instances:

OSFldAss: $\langle \Gamma, \sigma, \rho, \lambda, \iota\texttt{.}f \texttt{ = } u\rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \mathsf{fldUpd}(\sigma, \iota, f, u), \rho, \lambda, u\rangle$

We conclude our discussion of the operational semantics with the two circumstances in which variables are scoped—method call, and the `for` iterator.

The semantics for method call is given in OSCall. Access to the formal parameter, x, local variables, $x_{1..i}$, and `this` must be scoped within the body of m, so we freshen these syntactic names to x', $x'_{1..i}$ and $x'_{\texttt{this}}$ in the style of Drossopoulou et al. [7].

OSCall: $\langle \Gamma_1, \sigma, \rho, \lambda_1, \iota\texttt{.}m\texttt{(}u\texttt{)}\rangle \overset{P}{\rightsquigarrow} \langle \Gamma_2, \sigma, \rho, \lambda_2, \texttt{\{ } s_2 \texttt{ return } e_2\texttt{; \}}\rangle$
where
$\sigma(\iota) = \langle\!\langle n \| \ldots \rangle\!\rangle$ or $\sigma(\iota) = \langle\!\langle n, \_, \_, \_ \| \ldots \rangle\!\rangle$
$\mathcal{MD}_{P,n}(m) = (x, \mathcal{L}, t_1, \_, s_1 \texttt{ return } e_1\texttt{;})$
$\mathsf{dom}(\mathcal{L}) = \{x_1, \ldots, x_i\}$
$x', x'_{\texttt{this}}, x'_1, \ldots, x'_i \notin \mathsf{dom}(\lambda_1)$
$\Gamma_2 = \Gamma_1[x' \mapsto t_1][x'_{\texttt{this}} \mapsto n][x'_{1..i} \mapsto \mathcal{L}(x_{1..i})]$
$\lambda_2 = \lambda_1[x' \mapsto u][x'_{\texttt{this}} \mapsto \iota][x'_{1..i} \mapsto \mathsf{initial}(\Gamma_2(x'_{1..i}))]$
$s_2 = s_1[x'/x][x'_{1..i}/x'_{1..i}][x'_{\texttt{this}}/\texttt{this}]$
$e_2 = e_1[x'/x][x'_{1..i}/x_{1..i}][x'_{\texttt{this}}/\texttt{this}]$

We extend the typing environment, Γ_2, with new local variable type bindings for the fresh names (as well as those for the formal parameter and `this`), and include appropriate initial values in the locals store, λ_2. Finally, the old syntactic names are updated in the method body, s, and `return` expression, e, by substitution.

A similar strategy is used to avoid binding clashes for the `for` iterator:

OSFor1: $\langle \Gamma, \sigma, \rho, \lambda, \texttt{for (}n\, x \texttt{ : } \emptyset\texttt{) \{}s_1\texttt{\}; } s_2\rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, s_2\rangle$

OSFor2: $\langle \Gamma_1, \sigma, \rho, \lambda_1, \texttt{for (}n\ x\ \texttt{:}\ u\texttt{)}\ \{s_1\}\texttt{;}\ s_2\rangle \overset{P}{\rightsquigarrow}$
$\langle \Gamma_2, \sigma, \rho, \lambda_2, s_3\ \texttt{for (}n\ x\ \texttt{:}\ (u \setminus \iota)\texttt{)}\ \{s_1\}\texttt{;}\ s_2\rangle$
where
$\iota \in u, x \neq x' \notin \mathsf{dom}(\lambda_1)$
$\Gamma_2 = \Gamma_1[x' \mapsto x], \lambda_2 = \lambda_1[x' \mapsto \iota], s_3 = s_1[x'/x]$

Iteration of the empty set evaluates immediately to 'skip', while iteration over the non-empty set picks an element from the set, assigns this to the iterator variable, and unfolds the statement block, in which the bound iterator variable is freshened. We do not specify the order in which the elements of u are bound to x.

5 Soundness

In this section we outline proofs of two key safety properties: that no type-correct program will get 'stuck'—except in a well-defined error state—and that types are preserved during program execution.

Firstly, however, we define some well-formedness properties of stores and values, so that we can check type preservation through subject reduction.

Value Typing and Well-Formedness
We redefine our typing relation to include the store, σ, so that values may be typed—particularly important for showing subject-reduction. Typings of `true` and `false` with `boolean`, and of `null` with any valid nominal type are elided.

Firstly, an address has a type, n, if the object at that address in the store has a dynamic type (written $\mathsf{dynType}(\sigma(\iota))$) subordinate to n. This condition is then mapped over the members of a set of addresses in DTSet:

$$\text{(DTAddr)}\ \frac{\vdash \mathsf{dynType}(\sigma(\iota)) \leq n}{P, \Gamma, \sigma \vdash \iota : n} \qquad \text{(DTSet)}\ \frac{\begin{array}{c} P \vdash n \\ \forall j \in 1..i : P, \Gamma, \sigma \vdash \iota_j : n \end{array}}{P, \Gamma, \sigma \vdash \{\iota_1, \ldots, \iota_i\} : \texttt{set<}n\texttt{>}}$$

We also provide a typing rule for the method body construction introduced in Fig. 5:

$$\text{(DTMethBody)}\ \frac{\begin{array}{c} P, \Gamma, \sigma \vdash s \\ P, \Gamma, \sigma \vdash e : t \end{array}}{P, \Gamma, \sigma \vdash \{\ s\ \texttt{return}\ e\texttt{;}\ \} : t}$$

We make use of a 'well-formed object' relation, $P, \sigma \vdash o \diamond_{\mathsf{inst}}$, when o is a well-formed object in some store, the rules for which follow:

$$\text{(WFField)}\ \frac{\begin{array}{c} \mathsf{dynType}(o) = n \\ \mathcal{FD}_{P,n}(f) = t \\ P, \emptyset, \sigma \vdash o(f) : t \end{array}}{P, \sigma, o \vdash f \diamond_{\mathsf{fld}}}$$

WFField checks that the field f stores a value of appropriate type for its definition in class or relationship n, according the dynamic typing relation given above. This relation is mapped across the fields of classes and relationships in the following rules:

$$\text{(WFObject1)}\quad \frac{}{P, \sigma \vdash \langle\!\langle \texttt{Object} \| \rangle\!\rangle \diamond_{\mathsf{inst}}} \qquad \text{(WFRelInst1)}\quad \frac{\iota_1, \iota_2 \in \mathsf{dom}(\sigma)}{P, \sigma \vdash \langle\!\langle \texttt{Relation}, \texttt{null}, \iota_1, \iota_2 \| \rangle\!\rangle \diamond_{\mathsf{inst}}}$$

$$\text{(WFObject2)}\quad \frac{\begin{array}{c}\{f_1, \ldots, f_i\} = \mathsf{dom}(\mathcal{FD}_{P,c}) \\ \forall j \in 1..i : P, \sigma, o \vdash f_j \diamond_{\mathsf{fld}}\end{array}}{P, \sigma \vdash \langle\!\langle c \| f_1 : v_1, \ldots, f_i : v_i \rangle\!\rangle \diamond_{\mathsf{inst}}} \qquad \text{(WFRelInst2)}\quad \frac{\begin{array}{c}\mathcal{R}_P(r) = (\mathsf{dynType}(\sigma(\iota)), n_1, n_2, \mathcal{F}, \_) \\ \{f_1, \ldots, f_i\} = \mathsf{dom}(\mathcal{F}) \\ \forall j \in 1..i : P, \sigma, o \vdash f_j \diamond_{\mathsf{fld}} \\ \vdash \mathsf{dynType}(\sigma(\iota_1)) \leq n_1 \\ \vdash \mathsf{dynType}(\sigma(\iota_2)) \leq n_2\end{array}}{P, \sigma \vdash \langle\!\langle r, \iota, \iota_1, \iota_2 \| f_1 : v_1, \ldots, f_i : v_i \rangle\!\rangle \diamond_{\mathsf{inst}}}$$

WFObject1 and WFRelInst1 specify that instances of `Object` and `Relation`, respectively, are valid. WFObject2, requires that all fields are well-formed and that the class instance has precisely those fields that were declared or inherited. WFRelInst2, checks that only those fields *immediately* declared in r are present in the relationship instance; that those fields are well-formed; that the super-instance, at ι, is present, and has a dynamic type equal to r's supertype; and that the r-instance sits between two instances of appropriate type according to r's definition.

We check that the relationships are properly specified in ρ according to the following two rules:

$$\text{(WFRelation1)}\quad \frac{\sigma(\rho(\texttt{Relation}, \iota_1, \iota_2)) = \langle\!\langle \texttt{Relation}, \texttt{null}, \iota_1, \iota_2 \| \rangle\!\rangle}{P, \sigma, \rho \vdash (\texttt{Relation}, \iota_1, \iota_2) \diamond_{\mathsf{rel}}}$$

$$\text{(WFRelation2)}\quad \frac{\begin{array}{c}\mathcal{R}_P(r) = (r', \_, \_, \_, \_) \\ (r', \iota_1, \iota_2) \in \mathsf{dom}(\rho) \\ \sigma(\rho(r, \iota_1, \iota_2)) = \langle\!\langle r, \rho(r', \iota_1, \iota_2), \iota_1, \iota_2 \| \ldots \rangle\!\rangle\end{array}}{P, \sigma, \rho \vdash (r, \iota_1, \iota_2) \diamond_{\mathsf{rel}}}$$

WFRelation2 ensures that the r-instance between ι_1 and ι_2 has a super-instance that also sits between ι_1 and ι_2. WFRelation1 acts as a base-case for `Relation`, instances of which do not take a super-instance.

We then map the conditions for well-formed instances, relations and local variables over the heap, σ, the relationship heap, ρ, and the locals map, λ:

$$\text{(WFHeap)}\quad \frac{\forall \iota \in \mathsf{dom}(\sigma) : P, \sigma \vdash \sigma(\iota) \diamond_{\mathsf{inst}}}{P \vdash \sigma \diamond_{\mathsf{heap}}} \qquad \text{(WFRelHeap)}\quad \frac{\forall (r, \iota_1, \iota_2) \in \mathsf{dom}(\rho) : P, \sigma, \rho \vdash (r, \iota_1, \iota_2) \diamond_{\mathsf{rel}}}{P, \sigma \vdash \rho \diamond_{\mathsf{relheap}}}$$

$$\text{(WFLocals)}\quad \frac{\forall x \in \mathsf{dom}(\Gamma) : P, \Gamma, \sigma \vdash \lambda(x) : \Gamma(x)}{P, \Gamma, \sigma \vdash \lambda \diamond_{\mathsf{locals}}}$$

We consider a configuration $\langle \Gamma, \sigma, \rho, \lambda, s \rangle$ to be well-formed when σ, ρ and λ are well-formed, and where s is type-correct. Error configurations, $\langle \Gamma, \sigma, \rho, \lambda, w \rangle$, are well-formed under similar conditions.

Safety

Type safety is shown by a subject reduction theorem, central to which is the idea that context substitution respects types:

Lemma 1 (Substitution). *For expressions e_1 and e_2, which are typed t_1 and t_2 respectively, where t_2 is a subtype of t_1 and where $\mathcal{E}_{\mathrm{e}}[e_1]$ is typed t_3, then $\mathcal{E}_{\mathrm{e}}[e_2]$ has a subtype of t_3.*

The proof follows by induction on the structure of the typing derivation. Next, we show type preservation, which follows naturally from the previous lemma, and by induction on the structure of the derivation of execution:

Theorem 1 (Subject Reduction). *In a well-typed program, P, where $\langle \Gamma_1, \sigma_1, \rho_1, \lambda_1, s_1 \rangle$ executes to a new configuration $\langle \Gamma_2, \sigma_2, \rho_2, \lambda_2, s_2 \rangle$, that configuration will be well-formed. Furthermore, $\Gamma_1 \subseteq \Gamma_2$ and all objects in σ_1 retain their dynamic type in σ_2.*

Similarly where the original configuration executes to an error configuration.

Finally, we show that a well-typed program may always perform an execution step:

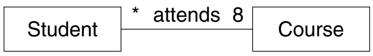
Theorem 2 (Progress). *For all well-typed programs, P, all well-formed configurations $\langle \Gamma_1, \sigma_1, \rho_1, \lambda_1, s_1 \rangle$ execute to either:*

i. an error configuration $\langle \Gamma_2, \sigma_2, \rho_2, \lambda_2, w \rangle$, or
ii. a new statement configuration $\langle \Gamma_2, \sigma_2, \rho_2, \lambda_2, s_2 \rangle$

By Theorems 1 and 2, any well-typed program can make a step to a new well-formed configuration: well-typed programs do not go wrong.

6 Restricting Multiplicities

In UML, associations can be annotated with *multiplicities*, which restrict the number of instances that may take part in any given relation. For example, it could be that every student attends exactly eight courses, but that a course may have any number of students:



More exotic multiplicities can include ranges ('1..7'), and comma-separated ranges ('1..7, 10..*'). There are a number of ways in which such restrictions could be expressed in RelJ. We describe below both a flexible, but dynamically checked approach, as well as a more restricted, statically checked approach.

6.1 Dynamic Approach

The use of a run-time check at every relationship addition would allow us to represent most of the possible multiplicities that can be expressed in UML. When, say, too many courses are added to the `Attends` relationship, an exception could be raised:

```
relationship Attends (many Student, 2 Course) { int mark; }
...
Attends.add(alice, programming);
Attends.add(alice, semantics);
Attends.add(alice, types);       // Exception!
```

We deviate from UML slightly: an association annotated at one end with ‘2’ would always have exactly two associated instances. Instead, we interpret our `2` annotation on `Course` as ‘0..2’ in UML notation: that is, courses start without any students.

6.2 Static Approach

Our preference, however, is for a static approach to the expression of multiplicities. While less flexible, we need not generate constraint-checking code for relationship additions, and we provide more robust guarantees that the multiplicity constraints are satisfied. Rather than give the formal details, we shall give an overview of this extension to RelJ.

We only allow `one` and `many` annotations. The former is equivalent to ‘0..1’ in UML, the latter to ‘0..*’:

```
relationship Attends (many Student, many Course);
relationship Failed (many PassedStudent, one Course);
```

In the declarations above, we see that students’ course attendance is unrestricted, but that a `PassedStudent` may have failed at most one course.

We further restrict relationship inheritance so that a many-to-one relationship may only inherit from a many-to-one or many-to-many relationship. We impose similar restrictions on many-to-many and one-to-many relationship definitions. We then add to the invariants of Sect. 2.

Invariant 3. *For a relationship r, declared “`relationship` r `(`n_1`,` n_2`)`”, where n_1 is annotated with `one`, there is at most one n_1-instance related through r to every n_2-instance. The converse is true where n_2 is annotated with `one`.*

There is a tension between Invariants 1 and 3. Consider the following relationship definitions, where a course can only be taught by a single lecturer, and where lecturers enjoy teaching hard courses, but teach them slowly:

```
relationship Teaches (one Lecturer, many Course);
relationship ExcitedlyTeaches extends Teaches
                            (one Lecturer, many HardCourse);
```

```
relationship SlowlyTeaches extends Teaches
                            (one Lecturer, many HardCourse);

charlie = new Lecturer();
deirdre = new Lecturer();
advancedWidgets = new HardCourse();
```

Suppose that `charlie ExcitedlyTeaches advancedWidgets`, then by Invariant 1, `charlie` also `Teaches advancedWidgets`.

Now suppose that `deirdre` is to slowly teach `advancedWidgets`:

```
SlowlyTeaches.add(deirdre, advancedWidgets);
```

By Invariant 1, `deirdre` must also be related to `advancedWidgets` via `Teaches`. However, by Invariant 3, `charlie` and `deirdre` cannot *both* `Teach advancedWidgets`. In our formalised semantics, we remove `charlie` from `Teaches` with `advancedWidgets`: the `add` becomes an assignment, rather than an addition, in this case. Furthermore, by Invariant 1, `charlie` cannot be in `ExcitedlyTeaches` with `advancedWidgets` once he has been removed from `Teaches`—therefore, he is also removed from `ExcitedlyTeaches`.

This behaviour, where not only sub-relationships of r are altered by a change to r's contents, but possibly also the contents of parents and siblings of r, might seem unexpected. At the same time, they make sense when examining examples, and provide a means for avoiding run-time checks.

7 Conclusion

In this paper, we have presented RelJ, a core fragment of Java that offers first-class support for first-class relationships. Unlike other work, we have formally specified our language; giving mathematical definitions of its type system and operational semantics. Given such definitions we are able prove an important correctness property of our language.

7.1 Related Work

Modelling languages like UML [9] and ER-diagrams [5] provide associations and relationships as core abstractions. Several database systems, for example object databases adhering to the ODMG standard [12], also provide relationships as primitives. Unfortunately, programming languages provide no first-class access to such primitives, so weak APIs must be used instead.

As we mentioned earlier, Rumbaugh [13] was the first to point out that relationships have an important rôle to play in general object-oriented languages, and gave an informal description of a language based on Smalltalk. However, the matter of relationship inheritance was mentioned only as an analogue to class inheritance, and there was no formal treatment of this or the language as a whole.

Noble has presented some patterns for programming with relationships [10]. In fact, many of these patterns could be used in translating RelJ programs to 'pure' Java. Noble and Grundy also suggested that relationships should be made explicit in object-oriented programs [11]. Again, neither work provides any concrete details of language support for relationships.

After completing the first draft of this work we discovered the paper by Albano, Ghelli and Orsini [1], which describes a language based on associations (relationships) for use in an object-oriented database environment. Their data model is quite different from ours; for example, they treat classes as containers, or extents [12]. Thus values can inhabit multiple classes, and classes also support multiple inheritance. In fact, classes turn out to be unary associations, which is the core abstraction in Albano et al.'s model.

Their model also provides a rich range of constraints; for example, surjectivity and cardinality constraints for associations, and disjointness constraints on classes. These are compiled to the appropriate runtime checks. (They take advantage of the underlying database infrastructure and utilize triggers and transactions.) Finally, they give no formal description of the language.

Our work, in contrast, takes as its starting point the Java object model and hence much of the complexity of Albano et al.'s model is simply not available. However, a notion of 'container' can be easily coded up. First assume a class `Singleton` and a single object of that class, called `default`. We can then define containers for the `Person` and `Student` classes of Fig. 2 as follows (where we assume a super-relationship `Extent` between `Singleton` and `Object` classes).

```
relationship Persons extends Extent
                        (Singleton, Person) {
}
relationship Students extends Persons
                        (Singleton, Student) {
}
```

So to place `Tom` in the `Persons` container we simply write `Persons.add(default, Tom)`. Similarly `Students.add(default, Jerry)` would add the object `Jerry` to the `Students` container, and by delegation also in the `Persons` container. The expression `default.Persons` would return the current contents of the `Persons` container. (Syntactic sugar could easily be added to make this code a little more compact.)

Interest in relationships is not restricted to modelling and programming languages. In the timeframe of the next generation of Microsoft Windows, codenamed 'Longhorn', the Windows storage subsystem will be replaced with a new system called *WinFS*. WinFS provides a database-like file store, the core of which is a collection of *items*, like objects, which represent data such as images, Outlook contacts, and user-defined items. The other key component of the WinFS data model is relationships, which are defined between items. WinFS thus represents a move away from the traditional tree-based file system hierarchy to an arbitrary graph-based file system, where the key abstraction is the relationship. At the time of writing, details of the API for WinFS are scarce, but it is clear that a language such as RelJ would provide a more direct programming framework, where various compile-time checks and optimizations would be possible. When the details of WinFS are finalized and made public, it would be interesting to compare various systems routines written in a language such as RelJ with those written using the APIs.

7.2 Further Work

Clearly RelJ is just a first step in providing comprehensive first-class support of relationships in an object-oriented language. There are several features available in modelling languages, such as UML, that cannot currently be expressed in RelJ; notably, we only support relationships that are one-way. We hope to add relationships that may be traversed in both directions safely, as well as further investigating multiplicities.

In this paper we have not given details of how RelJ can be implemented. To support it directly in the runtime would require considerable extension of the JVM. The design and evaluation of such an extension is interesting future work. As an alternative, we have informally specified a systematic translation of RelJ into 'pure' Java. In the future, we plan to formalize this translation and prove it correct.

Another direction we wish to consider is extending RelJ with more query-like facilities (in a style similar to Cω [3]). For example, one might add a simple filter facility, e.g. the expression `alice.Attends[it.title.matches("*101")]` would return the beginners' courses that `alice` is currently attending. (The subexpression in square brackets is a simple boolean-valued expression, where `it` is bound to each element of the relationship in turn.)

Finally, we conclude by recording our hope that our language may provide a first step in the process of principled unification of modelling languages (UML, ER-diagrams), programming languages (Java, C$^\sharp$), and data query and specification languages (SQL, schema design).

Acknowledgments

Much of this work was completed whilst Bierman was at the University of Cambridge Computer Laboratory and supported by EU grants Appsem-II and EC FET-GC project IST-2001-33234 Pepito. Wren is currently supported by an EPSRC studentship. We are grateful to Sophia Drossopoulou and her group for useful comments on this work, as well as to Matthew Fairbairn, Giorgio Ghelli, Alan Mycroft, James Noble, Matthew Parkinson, Andrew Pitts, Peter Sewell and the attendees of FOOL 2005.

References

1. A. Albano, G. Ghelli, and R. Orsini. A relationship mechanism for a strongly typed object-oriented database programming language. In *Proceedings of VLDB*, 1991.
2. C. Anderson and S. Drossopoulou. δ: An imperative object-based calculus with delegation. In *Proceedings of USE*, 2002.
3. G. Bierman, E. Meijer, and W. Schulte. The essence of Cω. In *Proceedings of ECOOP*, 2005.

4. G. Bierman, M. Parkinson, and A. Pitts. MJ: A core imperative calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory, 2003.
5. P. P.-S. Chen. The entity-relationship model – toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.
6. S. Drossopoulou. An abstract model of Java dynamic linking and loading. In *Proceedings of Types in Compilation (TIC)*, 2000.
7. S. Drossopoulou, T. Valkevych, and S. Eisenbach. Java type soundness revisited, September 2000.
8. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of POPL*, pages 171–183, 1998.
9. I. Jacobson, G. Booch, and J. Rumbaugh. *The unified software development process.* Addison-Wesley, 1999.
10. J. Noble. Basic relationship patterns. In *Pattern Languages of Program Design, vol. 4.* Addison Wesley, 1999.
11. J. Noble and J. Grundy. Explicit relationships in object-oriented development. In *Proceedings of TOOLS*, 1995.
12. R.G.G. Cattell et al. *The Object Data Standard: ODMG 3.0.* Morgan Kaufmann, 2000.
13. J. Rumbaugh. Relations as semantic constructs in an object-oriented language. In *Proceedings of OOPSLA*, pages 466–481, 1987.
14. J. Smith and D. Smith. Database abstractions: Aggregation and generalizations. *ACM Transactions on Database Systems*, 2(2):105–133, 1977.
15. P. Stevens and R. Pooley. *Using UML: software engineering with objects and components.* Addison-Wesley, 1999.
16. D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings of OOPSLA*, pages 227–242. ACM Press, 1987.
17. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

A Details of Type System and Semantics

This appendix contains the details of the semantics not covered in the main body of the paper.

A.1 Typing Rules

In addition to the subtyping rules given in Sect. 3, the following rules populate the subtyping relation with the immediate supertypes provided by the language syntax, and give the reflexive, transitive closure:

$$\begin{array}{cccc}
\text{(STRef)} & \text{(STTrans)} & \text{(STClass)} & \text{(STRel)} \\
\dfrac{P \vdash t}{\vdash t \leq t} & \dfrac{\vdash t_1 \leq t_2 \quad \vdash t_2 \leq t_3}{\vdash t_1 \leq t_3} & \dfrac{\mathcal{C}(c_1) = (c_2, \_, \_)}{\vdash c_1 \leq c_2} & \dfrac{\mathcal{R}(r_1) = (r_2, \_, \_, \_, \_)}{\vdash r_1 \leq r_2}
\end{array}$$

The typing rules for the RelJ statements and expressions not typed in Sect. 3 are shown in Fig. 7.

We omit the typing of literal values `true`, `false`, `null` and `empty`, which are typed in the obvious way – `boolean`, n and `set<`n`>` respectively. Variables

$$\text{(TSVar)}\ \frac{\Gamma(x) = t}{\Gamma \vdash x : t} \qquad \text{(TSNew)}\ \frac{P \vdash c}{\Gamma \vdash \texttt{new}\ c\texttt{()} : c} \qquad \text{(TSEq)}\ \frac{\Gamma \vdash e_1 : n \quad \Gamma \vdash e_2 : n'}{\Gamma \vdash e_1\ \texttt{==}\ e_2 : \texttt{boolean}} \qquad \text{(TSFld)}\ \frac{\Gamma \vdash e : n \quad \mathcal{FD}_n(f) = t}{\Gamma \vdash e\texttt{.}f : t}$$

$$\text{(TSAdd)}\ \frac{\Gamma \vdash e_1 : \texttt{set<}n_1\texttt{>} \quad \Gamma \vdash e_2 : n_2 \quad \vdash n_1 \leq n_3 \quad \vdash n_2 \leq n_3}{\Gamma \vdash e_1\ \texttt{+}\ e_2 : \texttt{set<}n_3\texttt{>}} \qquad \text{(TSSub)}\ \frac{\Gamma \vdash e_1 : \texttt{set<}n_1\texttt{>} \quad \Gamma \vdash e_2 : n_2 \quad \vdash n_1 \leq n_3 \quad \vdash n_2 \leq n_3}{\Gamma \vdash e_1\ \texttt{-}\ e_2 : \texttt{set<}n_3\texttt{>}}$$

$$\text{(TSAss)}\ \frac{x \neq \texttt{this} \quad \Gamma \vdash x : t_1 \quad \Gamma \vdash e : t_2 \quad \vdash t_2 \leq t_1}{\Gamma \vdash x\ \texttt{=}\ e : t_2}$$

$$\text{(TSFldAss)}\ \frac{\Gamma \vdash e_1 : n \quad \Gamma \vdash e_2 : t_1 \quad \mathcal{FD}_n(f) = t_2 \quad \vdash t_1 \leq t_2}{\Gamma \vdash e_1\texttt{.}f\ \texttt{=}\ e_2 : t_1} \qquad \text{(TSCall)}\ \frac{\Gamma \vdash e_1 : n \quad \Gamma \vdash e_2 : t_1 \quad \mathcal{MD}_n(m) = (x, \mathcal{L}, t_2, t_3, \_) \quad \vdash t_1 \leq t_2}{\Gamma \vdash e_1\texttt{.}m\texttt{(}e_2\texttt{)} : t_3}$$

$$\text{(TSCond)}\ \frac{\Gamma \vdash e : \texttt{boolean} \quad \Gamma \vdash s_1 \quad \Gamma \vdash s_2 \quad \Gamma \vdash s_3}{\Gamma \vdash \texttt{if (}e\texttt{)}\ \{s_1\}\ \texttt{else}\ \{s_2\}\texttt{;}\ s_3}$$

$$\text{(TSSkip)}\ \frac{}{\Gamma \vdash \epsilon}$$

Fig. 7. The remaining type rules of RelJ

are typed by TSVar simply by look-up in the typing environment. Note that TSVar covers the type of `this` by its inclusion in VarName. New class-instance allocation is typed in the obvious way. The equality test is valid as long as both expressions are addresses. (Similar rules are required for e_1 and e_2 as `set<`$-$`>` or `boolean` types, but these are obvious and omitted.) Field look-up is typed from the field table of the receiver's static type. Rules TSVarAdd to TSFldSub demonstrate object addition and removal from set values. In all cases, the right-hand operand must be the address of an object with a type subordinate to the set's static type. The entire expression takes the right-hand operand's type. Variables and fields may be assigned values subordinate to the left-hand side's declared type. Method call is typed directly from the method look-up table. The `for` statement was typed in the body of the paper. The conditional's typing-checking is standard, recalling that we do not assign types to statements. All statements require that their continuation statement is also well-typed, and we explicitly type the empty statement (ϵ), which is usually omitted in program text.

Finally, a program is well-typed if all of its classes and relationships are well-typed, if classes and relationships are disjoint, and if the subtyping relationship is antisymmetric:

$$\text{(TSProgram)}\ \frac{\forall n \in \mathsf{dom}(\mathcal{C}_P) \cup \mathsf{dom}(\mathcal{R}_P) : P \vdash n \qquad \forall n_1, n_2 : P \vdash n_1 \leq n_2 \wedge P \vdash n_2 \leq n_1 \Rightarrow n_1 = n_2}{\vdash P}$$

A.2 Operational Semantics

First, we give full definitions of new, which returns an initialised class instance; $\mathsf{initial}$, which returns an appropriate initial value for a variable of type t; $\mathsf{dynType}$, which returns the dynamic type of an address in the store; and of fld, which returns the value of field f in the object at ι in store σ, delegating the field lookup to the superinstance as appropriate.

$$\mathsf{new}_P(c) = \begin{cases} \langle\!\langle \mathtt{Object} \| \rangle\!\rangle & \text{if } c = \mathtt{Object} \\ \langle\!\langle c \| f_1 : \mathsf{initial}_P(\mathcal{FD}_{P,c}(f_1)), \ldots, f_i : \mathsf{initial}_P(\mathcal{F}_{P,c}(f_i)) \rangle\!\rangle & \text{otherwise} \end{cases}$$
$$\text{where } \{f_1, f_2, \ldots, f_i\} = \mathsf{dom}(\mathcal{FD}_{P,c})$$

$$\mathsf{initial}_P(t) = \begin{cases} \mathtt{null} & \text{if } t = n' \\ \mathtt{false} & \text{if } t = \mathtt{boolean} \\ \emptyset & \text{if } t = \mathtt{set<}n\mathtt{>} \end{cases}$$

$$\mathsf{dynType}(o) = n \text{ where } o = \langle\!\langle n \| \ldots \rangle\!\rangle \ \vee \ o = \langle\!\langle n, \_, \_, \_ \| \ldots \rangle\!\rangle$$

$$\mathsf{fld}(\sigma, f, \iota) = \begin{cases} \sigma(\iota)(f) & \text{if } f \in \mathsf{dom}(\sigma(\iota)) \text{ or} \\ \mathsf{fld}(\sigma, f, \iota') & \text{if } f \notin \mathsf{dom}(\sigma(\iota)) \ \wedge \ \sigma(\iota) = \langle\!\langle r, \iota', \_, \_ \| \ldots \rangle\!\rangle \end{cases}$$

The remaining rules of the operation semantics are then as follows:

OSEmpty: $\langle \Gamma, \sigma, \rho, \lambda, \mathtt{empty} \rangle \stackrel{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \emptyset \rangle$

OSVar: $\langle \Gamma, \sigma, \rho, \lambda, x \rangle \stackrel{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \lambda(x) \rangle$

OSFldN: $\langle \Gamma, \sigma, \rho, \lambda, \mathtt{null}.f \rangle \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$

OSFld: $\langle \Gamma, \sigma, \rho, \lambda, \iota.f \rangle \stackrel{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{fld}(\sigma, \iota, f) \rangle$

OSRelInstN: $\langle \Gamma, \sigma, \rho, \lambda, \mathtt{null{:}}r \rangle \stackrel{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$

OSEq: $\langle \Gamma, \sigma, \rho, \lambda, u \ \mathtt{==} \ u \rangle \stackrel{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \mathtt{true} \rangle$

OSNeq: $\langle \Gamma, \sigma, \rho, \lambda, u \ \mathtt{==} \ u' \rangle \stackrel{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \mathtt{false} \rangle$ where $u \neq u'$

OSNew: $\langle \Gamma, \sigma, \rho, \lambda, \mathtt{new} \ c\mathtt{()} \rangle \stackrel{P}{\rightsquigarrow} \langle \Gamma, \sigma[\iota \mapsto \mathsf{new}_P(c)], \rho, \lambda, \iota \rangle$ where $\iota \notin \mathsf{dom}(\sigma)$

OSBody: $\langle \Gamma, \sigma, \rho, \lambda, \{ \ \mathtt{return} \ u\mathtt{;} \ \} \rangle \stackrel{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, u \rangle$

OSAdd: $\langle \Gamma, \sigma, \rho, \lambda, u \ \mathtt{+} \ \iota \rangle \stackrel{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, u \cup \{\iota\} \rangle$

OSAddN: $\langle \Gamma, \sigma, \rho, \lambda, u \ \mathtt{+} \ \mathtt{null} \rangle \stackrel{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$

OSSub: $\langle \Gamma, \sigma, \rho, \lambda, u \ \mathtt{-} \ \iota \rangle \stackrel{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, u \setminus \{\iota\} \rangle$

OSSubN: $\langle \Gamma, \sigma, \rho, \lambda, u \ \mathtt{-} \ \mathtt{null} \rangle \stackrel{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$

OSVarAss: $\langle \Gamma, \sigma, \rho, \lambda, x \ \mathtt{=} \ u \rangle \stackrel{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda[x \mapsto u], u \rangle$

OSFldAssN: $\langle \Gamma, \sigma, \rho, \lambda, \mathtt{null}.f \ \mathtt{=} \ u \rangle \stackrel{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$

OSRelAddN: $\langle \Gamma, \sigma, \rho, \lambda, r\mathtt{.add(}\iota_1^{\mathtt{null}}\mathtt{,}\iota_2^{\mathtt{null}}\mathtt{)} \rangle \stackrel{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$
where $\iota_1^{\mathtt{null}} = \mathtt{null}$ or $\iota_2^{\mathtt{null}} = \mathtt{null}$

OSRelRemN: $\langle \Gamma, \sigma, \rho, \lambda, r\mathtt{.rem(}\iota_1^{\mathtt{null}}\mathtt{,}\iota_2^{\mathtt{null}}\mathtt{)} \rangle \stackrel{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$
where $\iota_1^{\mathtt{null}} = \mathtt{null}$ or $\iota_2^{\mathtt{null}} = \mathtt{null}$

OSCallN: $\langle \Gamma, \sigma, \rho, \lambda, \mathtt{null}.m(u) \rangle \stackrel{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$

OSStat: $\langle \Gamma, \sigma, \rho, \lambda, u\mathtt{;} \ s \rangle \stackrel{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, s \rangle$

OSCondT: $\langle \Gamma, \sigma, \rho, \lambda, \mathtt{if} \ \mathtt{(true)} \ \{s_1\} \ \mathtt{else} \ \{s_2\}\mathtt{;} \ s_3 \rangle \stackrel{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, s_1 \ s_3 \rangle$

OSCondF: $\langle \Gamma, \sigma, \rho, \lambda, \mathtt{if} \ \mathtt{(false)} \ \{s_1\} \ \mathtt{else} \ \{s_2\}\mathtt{;} \ s_3 \rangle \stackrel{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, s_2 \ s_3 \rangle$

The Essence of Data Access in Cω

The Power is in the Dot!

Gavin Bierman[1], Erik Meijer[2], and Wolfram Schulte[3]

[1] Microsoft Research, UK
gmb@microsoft.com
[2] Microsoft Corporation, USA
emeijer@microsoft.com
[3] Microsoft Research, USA
schulte@microsoft.com

Abstract. In this paper we describe the data access features of Cω, an experimental programming language based on C$^\sharp$ currently under development at Microsoft Research. Cω targets distributed, data-intensive applications and accordingly extends C$^\sharp$'s support of both data and control. In the data dimension it provides a type-theoretic integration of the three prevalent data models, namely the object, relational, and semi-structured models of data. In the control dimension Cω provides elegant primitives for asynchronous communication. In this paper we concentrate on the data dimension. Our aim is to describe the *essence* of these extensions; by which we mean we identify, exemplify and formalize their essential features. Our tool is a small core language, FCω, which is a valid subset of the full Cω language. Using this core language we are able to formalize both the type system and the operational semantics of the data access fragment of Cω.

1 Introduction

Programming languages, like living organisms, need to continuously evolve in response to their changing environment. These evolutionary steps are typically quite modest: most commonly the provision of better or reorganized APIs. Occasionally a more radical evolutionary step is taken. One such example is the addition of generic classes to both Java [6] and C$^\sharp$[25].

We should like to argue that the time has come for another large evolutionary step to be taken. Much software is now intended for distributed, web-based scenarios. It is typically structured using a three-tier model consisting of a *middle tier* containing the business logic that extracts relational data from a *data services tier* (a database) and processes it to produce semi-structured data (typically XML) to be displayed in the *user interface tier*.

It is the writing of these middle tier applications that we should like to address. These applications are most commonly written in an object-oriented language such as Java or C$^\sharp$ and have to deal with relational data (essentially SQL tables), object graphs, and semi-structured data (XML, HTML).

In addition, these applications are fundamentally concurrent. Because of the inherent latency in network communication, the more natural model of concurrency is asynchronous. Accordingly, Cω provides a simple model of asynchronous (one-way) concurrency based on the join calculus [12]. For the rest of this paper, we shall focus exclusively on the data access aspects of Cω; the concurrency primitives have been discussed elsewhere [3]. Thus when we write Cω, we mean the language excluding the concurrency primitives.

Unfortunately common programming practice, and native API support for data access (e.g. JDBC and ADO.NET) leave a lot to be desired. For example, consider the following fragment taken (and mildly adapted) from the JDBC tutorial to query a SQL database (a user-supplied country is stored in variable `input`).

```
Connection con = DriverManager.getConnection(...);
Statement stmt = con.createConnection();
String query = "SELECT * FROM COFFEES WHERE Country='"+input+"'";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
  String s = rs.getString("Cof_Name");
  float n = rs.getFloat("Price");
  System.out.println(s+" - "+n);
}
```

Using strings to represent SQL queries is not only clumsy but also removes any possibility for static checking. The impedance mismatch between the language and the relational data is quite striking; e.g. a value is projected out of a row by passing a string denoting the column name and using the appropriate conversion function. Perhaps most seriously, the passing of queries as strings is often a security risk (the ‘‘script code injection” problem—e.g. consider the case when the variable `input` is the string `"' OR 1=1; --"`) [17].

Unfortunately API support in both Java and C$\sharp$ for XML and XPath/XQuery is depressingly similar (even those APIs that map XML values tightly to an object representation still offer querying facilities by string passing).

Our contention is that object-oriented languages need to evolve to support data access satisfactorily. This is hardly a new observation; a large number of academic languages have offered such support for both relational and semi-structured data (see, e.g. [1, 2, 20, 19, 15, 4]). In spite of the obvious advantages of these languages, it appears that their acceptance has been hampered by the fact that they are ‘‘different” from more mainstream application languages, such as Java and C$\sharp$. For example, HaskellDB [19] proposes extensions to the lazy functional language, Haskell; and TL [20] is a hybrid functional/imperative language with advanced type and module systems. We approach this language support problem from a different direction, which is to extend the common application languages themselves rather than creating another new language.

Closer to our approach is SQLJ [24]. This defines a way of embedding SQL commands directly in Java code. Moreover the results of SQL commands can be stored in Java variables and *vice versa*. Thus SQL commands are statically checked by the SQLJ compiler. SQLJ compilation consists of two stages; first to pre-process the embedded SQL, and second the ‘pure Java’ compilation. Thus the embedded SQL code is not part of the language *per se* (in fact all the embedded code is prefixed by the keyword `#sql`).

The chief difference is that Cω offers an integration of both the XML and relational data models with an object model.

Design Objectives of Cω. The aim of our project was to evolve an existing language, C$^\sharp$, to provide first-class support for the manipulation of relational and semi-structured data. (Although we have started with C$^\sharp$, our extensions apply equally well to other object-oriented languages, including Java.)

Addressing the title of our paper, the essence of the resulting language, Cω, is twofold: its extensions to the C$^\sharp$ type system and, perhaps more importantly, the elegant provision of query-like capabilities (the sub-title of our paper). Cω has been carefully designed around a set of core design principles.

1. Cω is a coherent extension of (the safe fragment of) C$^\sharp$, i.e. C$^\sharp$ programs should be valid Cω programs with the same behaviour.
2. The type system of Cω is intended to be both as simple as possible and closely aligned to the type system in the XPath/XQuery standard. Our intended users are C$^\sharp$ programmers who are familiar with XPath/XQuery.
3. From a programming perspective, the real power of Cω comes from its query-like capabilities. These have been achieved by generalizing member access to allow simple XPath-like path expressions.

Paper Organization. The rest of the paper is organized as follows. In §2 we give a comprehensive overview to the Cω programming language.[1] In §3.1 we identify and formalize FCω, a core fragment of Cω. In §3.2 we detail a simpler fragment, ICω, and in §3.4 show how FCω can be compiled to ICω. Using this compilation, we are able to show a number of properties of FCω in §3.5, including a type soundness theorem. We briefly discuss some related work in §4 and conclude in §5.

2 An Introduction to Cω

Our design goal was to evolve C$^\sharp$ to provide an integration of the object, relational and semi-structured data models. One possibility would be to add these data models to our programming language in an orthogonal way, e.g. by including new types `XML<`S`>` and `TABLE<`R`>`, where S and R are XML and relational schema respectively. We have sought to integrate these models by *generalization*, rather than by ad-hoc specializations. In the rest of this section we shall present the key ideas behind Cω, and give a number of small programs to illustrate these ideas. This section should serve as a programmer's introduction to Cω. We assume that the reader is familiar with C$^\sharp$/Java-like languages.

2.1 New Types

Cω is an extension of C$^\sharp$, so the familiar primitive types such as integers, booleans, floats are present, as well as classes and interfaces. In this section we shall consider

[1] An preliminary version of Cω was (informally) described in [22]. We have subsequently simplified the language, and our chief contribution here is a formalization (§§3–4).

in turn the extensions to the type system—streams, anonymous structs, discriminated unions, and content classes—and for each consider the new query capabilities.

Streams. The first structural type we add is a stream type; streams represent ordered homogeneous collections of zero or more values. For example, `int*` is the type for homogeneous sequences of integers. Streams in Cω are aligned with iterators, which will appear in C$^\sharp$ 2.0. Cω streams are typically generated using iterators, which are blocks that contain `yield` statements. For example, the `FromTo` method:

```
virtual int* FromTo(int b, int e){
  for (i = b; i <= e; i++) yield return i;
}
```

generates a finite, increasing stream of integers. Importantly, it should be noted that, just as for C$^\sharp$, invoking such a method body does *not* immediately execute the iterator block, but rather immediately returns a closure. (Thus Cω streams are essentially lazy lists, in the Haskell sense.) This closure is consumed by the `foreach` statement, e.g. the following code fragment builds a finite stream and then iterates over the elements, printing each one to the screen.

```
int* OneToHundred = FromTo(1,100);
foreach (int i in OneToHundred) Console.WriteLine(i);
```

A vital aspect of Cω streams is that they are always *flattened*; there are no nested streams of streams. Cω streams thus coincide with XPath/XQuery sequences which are also flattened. This alignment is a key design decision for Cω: it enables the semantics of our generalized member access to match the path selection of XQuery. We give further details later.

In addition, flattening of stream types also allows us to efficiently deal with recursively defined streams. Consider the following recursive variation of the function `FromTo` that we defined previously:

```
virtual int* FromTo2(int b, int e){
  if (b>e) yield break;
  yield return b;
  yield return FromTo2(b+1,e);
}
```

The statement `yield break;` returns the empty stream. The non-recursive call `yield return b` yields a single integer. The recursive call `yield return FromTo2 (b+1,n);` yields a stream of integers. As the type system treats the types `int*` and `int**` as equivalent this is type correct.

Without flattening we would be forced to copy the stream produced by the recursive invocation, leading to a quadratic instead of a linear number of `yields`:

```
virtual int* FromTo3(int b, int e){
  if (b>e) yield break;
  yield return b;
  foreach (int i in FromTo3(b+1,e))  yield return i;
}
```

Note that Cω's flattening of stream types does *not* imply that the underlying stream is flattened via some coercion; every element in a stream is `yield`-ed at most once. As we will see in the operational semantics (§3.3), iterating over a stream will effectively perform a depth-first traversal over the n-ary tree produced by the iterator.

Cω offers a limited but extremely useful form of *covariance* for streams. Covariance is allowed provided that the conversion on the element type is the identity; for example `Button*` is a subtype of `object*` whereas `int*` is *not* (as the conversion from `int` to `object` involves boxing). This notion is a simple variant of the notion of covariance for arrays in C$^\sharp$, although it is statically safe (unlike array covariance) as we can not overwrite elements of streams.

The rationale for this is that implicit conversions should be limited to constant-time operations. Coercing a stream of type `Button*` to type `object*` takes constant-time, whereas coercing `int*` to `object*` would be linear in the length of the stream, as the boxing conversion from `int` to `object` is not the identity.

A key programming feature of Cω is generalized member access; as the subtitle suggests the familiar 'dot' operator is now much more powerful. Thus if the receiver is a stream the member access is mapped over the elements, e.g. `OneToHundred.ToString()` implicitly maps the method call over the elements of the stream `OneToHundred` and returns a value of type `string*`. This feature significantly reduces the burden on the programmer. Moreover, member access has been generalized so it behaves like a *path expression*. For example, `OneToHundred.ToString().PadLeft(10)` converts all the elements of the stream `OneToHundred` to a string, and then pads each string, returning a stream of these padded strings.

Sometimes one wishes to map more than a simple member access over the elements of a stream. Cω offers a convenient shorthand called an *apply-to-all expression*, written e`.{`$\overline{s}$`}`, which applies the block `{`$\overline{s}$`}`, where $\overline{s}$ denotes a sequence of statements, to each element in the stream e.[2] The block may contain the variable `it` which plays a similar role as the implicit receiver argument `this` in a method body and is bound to each successive element of the iterated stream. (Such expressions are reminiscent of Smalltalk `do:` methods.) For example, the following code first creates the stream of natural numbers from 1 to 256, converts each of the elements to a hex string, converts each of these to upper case, and then applies an apply-to-all expression to print the elements to the screen:

```
FromTo(1,256).ToString("x").ToUpper().{ Console.WriteLine(it); };
```

Anonymous Structs. The second structural type we add are anonymous structs, which encapsulate heterogeneous ordered collections of values. An anonymous struct is like a tuple in ML or Haskell and is written as `struct{int i; Button;}` for example. A value of this type contains a member `i` of type `int` and an unlabelled member of type `Button`. We can construct a value of this type with the expression: `new{i=42, new Button()}`.

To access components of anonymous structs we (again) generalize the notion of member access. Thus assuming a value `x` of the previous type, we write `x.i` to ac-

[2] We shall adopt the FJ shorthand [18] and write $\overline{x}$ to mean a sequence of x.

cess the integer value. Unlabelled members are accessed by their position; for example `x[1]` returns the `Button` member. As for streams, member access is lifted over unlabelled members of anonymous structs. To access the `BackColor` property of the `Button` component in variable `x` we can just write `x.BackColor`, which is equivalent to `x[1].BackColor`.

At this point we can reveal even more of the power of Cω's generalized member access. Given a stream `friends` of type `struct{string name;int age;}*`, the expression `friends.age` returns a stream of integers. The member access is over *both* structural types. The following query-like statement prints the names of one's friends:

```
friends.name.{ ConsoleWriteLine(it);};
```

Interestingly, Cω also allows repeated occurrences of the same member name within an anonymous struct type, even at different types. For example, assume the following declaration: `struct{int i; Button; float i;} z;` Then `z.i` projects the two `i` members of `z` into a new anonymous struct that is equivalent to `new{z[0],z[2]}` and of type `struct{int;float;}`.

Cω provides a limited form of covariance for anonymous structs, just as for streams. For example, the anonymous struct `struct{int;Button;}` *is* a subtype of `struct{int; Control;}`. However it is *not* a subtype of `struct{object; Control;}` since the conversion from `int` to `object` is not an identity conversion. Cω does not support width subtyping for anonymous structs.

Choice Types. The third structural type we add is a particular form of discriminated union type, which we call a choice type. This is written, for example, `choice{int; bool;}`. As the name suggests, a value of this type is either an integer or a boolean, and may hold either at any one time. Unlike unions in C/C++ and variant records in Pascal where users have to keep track of which type is present, values of a discriminated union in Cω are implicitly tagged with the static type of the chosen alternative, much like unions in Algol68. In other words, discriminated union values are essentially a pair of a value and its static type.

There is no syntax for creating choice values; the injection is implicit (i.e. it is generated by the compiler).

```
choice{int;Button;} x = 3;
choice{int;Button;} y = new Button();
```

Cω provides a test, e `was` τ, on choice values to test the value's *static* type. Thus `x was int` would return `true`, whereas `y was int` would return `false`.

Assuming that an expression e is of type `choice{`$\overline{\tau}$`}`, the expression e `was` τ is true for *exactly one* τ in $\overline{\tau}$. This invariant is maintained by the type system. The only slight complication arises from subtyping, e.g.

```
choice{Control; object;} z = new Button();
```

As `Button` is a subtype of both `Control` and `object`, which type tag is generated by the compiler? A choice type can be thought of as providing a *family* of overloaded constructor methods, one for each component type. Just as for standard object creation in Java/C$^\sharp$, the *best* constructor method is chosen. In the example above, clearly

`Control` is better than `object`. Thus `z was Control` returns `true`. The notion of "best" for Cω is the routine extension of that for C♯.

As the reader may have guessed, member access has also been generalized over discriminated unions. Here the behaviour of member access is less obvious, and has been designed to coincide with XPath. Consider a value `w` of type `choice{char; Button;}`. The member access `w.GetHashCode()` succeeds irrespective of whether the value is a character or a `Button` object. In this case the type of the expression `w.GetHashCode()` is `int`.

However the member may not be supported by all the possible component types, e.g. `w.BackColor`. Classic treatments of union types would probably consider this to be type incorrect [23–p.207]. However, Cω's choice types follow the semantics of XPath where, for example, the query `foo/bar` returns the `bar` nodes under the `foo` node if any exist, and *the empty sequence* if none exist. Thus in Cω, the expression `w.BackColor` is well-typed, and will return a value of type `Color?`. This is another new type in Cω and is a variant of the nullable type to appear in C♯ 2.0. A value of type `Color?` can be thought of as a singleton stream, thus it is either empty or contains a single `Color` value (when `w` contains a `Button`). Again, we emphasize that this behaviour precisely matches that of XPath.

Cω follows the design of C♯ in allowing all values to be boxed and hence all value types are a subtype of the supertype `object`. Thus both anonymous structs and choice types are considered to be subtypes of the class `object`.

Content Classes. To allow close integration with XSD and other XML schema languages, we have included the notion of a *content class* in Cω. A content class is a normal class that has a single *unlabelled* type that describes the content of that class, as opposed to the more familiar (named) fields. The following is a simple example of a content class.

```
class friend{
  struct{ string name; int age; };
  void incAge(){...}
}
```

Again we have generalized member access over content classes. Thus the expression `Bill.age` returns an integer, where `Bill` is a value of type `friend`.

From an XSD perspective, classes correspond to global element declarations, while the content type of classes correspond to complex types. Further comparisons with the XML data model are immediately below, but a more comprehensive study can be found elsewhere [21].

2.2 XML Programming

It should be clear that the new type structures of Cω are sufficient to model simple XML schema. For example, the following XSD schema

```
<element name="Address"><complexType><sequence>
  <choice>
    <element name="Street" type="string"/>
    <element name="POBox" type="int"/>
  </choice>
  <element name="City" type="string"/>
</sequence></complexType></element>
```

can be represented (somewhat more succinctly!) as the Cω content class declaration:

```
  class Address {
    struct{
      choice{ string Street; int POBox; };
      string City;
    };
  }
```

The full Cω language supports XML literals as syntactic sugar for serialized object graphs. For example, we can create an instance of the `Address` class above using the following literal:

```
  Address a = <Address>
                <Street>13 Elm St</Street>
                <City>Hollywood</City>
              </Address>;
```

The Cω compiler contains a validating XML parser that deserializes the above literal into normal constructor calls. XML literals can also contain typed holes, much as in XQuery, that allow us to embed expressions to compute part of the literal. This is especially convenient for generating streams.

The inclusion of XML literals and the semantics of the generalized member access mean that XQuery code can be almost directly written in Cω. For example, consider one of the XQuery Use Cases [9], that processes a bibliography file (assume that this is stored in variable `bs`) and for each book in the bibliography, lists the title and authors, grouped inside a `result` element. The suggested XQuery solution is as follows.

```
  for $b in $bs/book
    return <result>{$b/title}{$b/author}<result>
```

The Cω solution is almost identical:

```
  foreach (b in bs.book)
    yield return <result>{b.title}{b.author}</result>;
```

The full Cω language adds several more powerful query expressions to those discussed in this paper. For instance, filter expressions e`[`e'`]` return the elements in the stream e that satisfy the boolean expression e'. As labels can be duplicated in anonymous structs and discriminated unions, the full language also allows type-based selection. For example, given a value `x` of type `struct{ int a; struct{string a;};}` we can select only the `string` member `a` by writing `x.string::a`.

Transitive queries are also supported in the full Cω language: the expression e`...`τ`::`m selects all members m of type τ that are transitively reachable from e. Transitive queries are inspired by the XPath descendant axis.

2.3 Database Programming

Relational tables are merely streams of anonymous structs. For example, the relational table created with the SQL declaration:

```
CREATE TABLE Customer (name string, custid int);
```

can be represented in Cω: `struct{string name; int custid}* Customer;`

In addition to path-like queries, the full Cω language also supports familiar SQL expressions, including `select-from-where`, various joins and grouping operators. Perhaps more importantly, these statements can be used on *any* value of the appropriate type, whether that value resides in a database or in memory; hence, one can write SQL queries in Cω code that does not access a database! One of the XQuery use-cases [9] asks to list the title prices for each book that is sold by both booksellers `A` and `BN`. Using a `select` statement and XML-literals, this query can be written in Cω as the following expression:

```
select <book-with-prices>
           <title>{a.title}</title>
           <price-A>{a.price}</price-A>
           <price-BN>{bn.price}</price-BN>
         </book-with-prices>
from book a in A.book, book bn in BN.book
where a.title == bn.title
```

Note the use of XML placeholders `{a.title}` and `{bn.price}`: when this code is evaluated new titles and new prices are computed from the bindings of the `select-from-where` clause.

So far we have shown how we can query values using generalized member and SQL expressions, but as Cω is an imperative language, we also allow to perform updates. This paper, however, focuses on the type extensions and generalized member access only.

3 The Essence of Cω

In the rest of this paper we study formally the essence of Cω, by which we mean we identify its essential features. We adopt a formal, mathematical approach and define a core calculus, Featherweight Cω, or FCω for short, similar to core subsets of Java such as FJ [18], MJ [5] and ClassicJava [11]. This core calculus, whilst lightweight, offers a similar computational ‘‘feel” to the full Cω language: it supports the new type constructors and generalized member access. FCω is a completely valid subset of Cω in that every FCω program is literally an executable Cω program.

The rest of this section is organized as follows. In §3.1 we define the syntax and type system for FCω. Rather than give an operational semantics directly for FCω we prefer to first ‘‘compile out” some of its features, in particular generalized member access. This both greatly simplifies the resulting operational semantics and demonstrates that Cω’s features do not require extensive new machinery. Thus in §3.2 we define a target language, Inner Cω, or ICω, for this ‘‘compilation”. ICω is essentially the same language, but for a handful of new language constructs and a much simpler type system. In §3.3 we give an operational semantics for ICω programs. In §3.4 we specify the compilation of FCω programs into ICω programs. This translation is, on the whole, quite straightforward. We conclude the section in §3.5 by stating some properties of our calculi and the compilation. Most important is the type-soundness property for ICω. Space prevents us from providing any details of the proofs, but they are proved using standard techniques and are similar to analagous theorems for fragments of Java [18, 5].

3.1 A Core Calculus: FCω

Syntax An FCω program consists of one or more class declarations. Each class declaration defines zero or more methods and contains exactly one unlabelled type that we call the *content type*. (We can code up a conventional C$^\sharp$/Cω class declaration with a number of field declarations using an anonymous struct.) FCω follows C$^\sharp$ and requires methods to be explicitly marked as `virtual` or `override`. Given a program we assume that there is a unique designated method within the class declarations that serves as the entry point.

Program	$p ::=$	$\overline{cd}$
Class Definition	$cd ::=$	$\texttt{class}\ c\texttt{:}c\ \texttt{\{}\tau\texttt{;}\overline{md}\texttt{\}}$
Method Definition	$md ::=$	$\texttt{virtual}\ \tau\ m\texttt{(}\overline{\tau}\ \overline{x}\texttt{)\{}\overline{s}\texttt{\}}$
	$\mid$	$\texttt{override}\ \tau\ m\texttt{(}\overline{\tau}\ \overline{x}\texttt{)\{}\overline{s}\texttt{\}}$

FCω supports two main kinds of types: *value types* and *reference types*. As usual, the distinguished type `void` is used for methods that do not return anything; `null` is only used to type `null` references, as with C$^\sharp$. Value types include the base types `bool` and `int` and the structural types: anonymous structs and discriminated unions. Reference types are either class types or streams. As usual only reference types have object identity and are represented at runtime by references into the heap. We assume a designated special class `object`.

Types				
$\tau ::= \gamma$	Value types	**Reference Types**		
$\mid\ \rho$	Reference types	$\rho ::= c$	Classes	
$\mid\ \texttt{void}\ \mid \texttt{null}$	Void and null types	$\mid\ \sigma\texttt{*}$	Stream types	
Value Types		$\mid\ \sigma\texttt{?}$	Singleton stream type	
$\gamma ::= b$	Base types			
$\mid\ \texttt{struct\{}\overline{fd}\texttt{\}}$	Anonymous structs	**Field Definition**		
$\mid\ \texttt{choice\{}\overline{\kappa}\texttt{\}}$	Choice types	$fd ::= \tau\ f\,;$	Named member	
Base Types		$\mid\ \tau\,;$	Unnamed member	
$b ::= \texttt{bool}\ \mid \texttt{int}$				

We employ the shorthand κ and σ to denote any type *except* a choice type and stream type (singleton or non-singleton), respectively. As Cω flattens stream types, we have made the simplification to FCω of removing nested stream types altogether from the type grammar. We have also simplified FCω choice types so that the members are unlabelled and we also exclude (for simplification) nested choice types. These can be coded up in FCω using unlabelled anonymous structs.

FCω expressions, as for C$^\sharp$, are split into ordinary expressions and promotable expressions. Promotable expressions are expressions that can be used as statements. We assume a number of built-in primitive operators, such as `==`, `||` and `&&`. In the grammar we write $e \oplus e$, where $\oplus$ denotes an instance of one of these operators. We do not formalize these operators further as their meaning is clear.

Expression

$e ::= b \mid i$	Literals
$\mid\ e \oplus e$	Built-in operators
$\mid\ x$	Variable
$\mid$ `null`	Null
$\mid\ (\tau)\,e$	Cast
$\mid\ e$ `is` τ	Dynamic typecheck
$\mid\ e$ `was` κ	Static typecheck for choice values
$\mid$ `new` $\tau(e)$	Object creation
$\mid$ `new` $\{\overline{be}\}$	Anonymous struct creation
$\mid\ e.f$	Field access
$\mid\ e[i]$	Field access by position
$\mid\ pe$	Promotable expression

Promotable expression

$pe ::= x = e$	Variable assignment
$\mid\ e.m(\overline{e})$	Method invocation
$\mid\ e.\{e\}$	Apply-to-all

Binding expression

$be ::= f = e$	Named binding
$\mid\ e$	Unnamed binding

We have made a simplification in the interests of space to restrict apply-to-all expressions to contain an expression rather than a sequence of statements. This simplifies the typing rules, but as apply-to-all expressions can be coded using `foreach` loops it is not a serious restriction.

Statements in FCω are standard. As mentioned earlier we have adopted the `yield` statement that will appear in C$^\sharp$ 2.0 to generate streams.

Statement	$s ::=$ `;`	Skip
	$\mid\ pe$`;`	Promoted expression
	$\mid$ `if` $(e)\ s$ `else` s	Conditional
	$\mid\ \tau\ x = e$`;`	Variable declaration
	$\mid$ `return` e`;`	Return statement
	$\mid$ `return;`	Empty return
	$\mid$ `yield return` e`;`	Yield statement
	$\mid$ `yield break;`	End of stream
	$\mid$ `foreach` $(\sigma\ x$ `in` $e)\ s$	Foreach loop
	$\mid$ `while` $(e)\ s$	While loop
	$\mid\ \{\overline{s}\}$	Block

In what follows we assume that FCω programs are well-formed, e.g. no cyclic class hierarchies, correct method body construction, etc. These conditions can be easily formalized but we suppress the details for lack of space.

Subtyping. Before we define the typing judgements for FCω programs we need to define a number of auxiliary relations. First we define the subtyping relation. We write $\tau <: \tau'$ to mean that type τ is a subtype of type τ'. The rules defining this relation are as follows.

$$\frac{}{\tau <: \tau}\,[\mathsf{Refl}] \quad \frac{\tau <: \tau' \qquad \tau' <: \tau''}{\tau <: \tau''}\,[\mathsf{Trans}] \quad \frac{}{\gamma <: \texttt{object}}\,[\mathsf{Box}] \quad \frac{\texttt{class}\ c : c'}{c <: c'}\,[\mathsf{Sub}]$$

$$\frac{}{\texttt{null} <: \rho}\,[\mathsf{Null}] \quad \frac{\tau <: \tau' \qquad f = f'}{\tau\, f <: \tau'\, f'}\,[\mathsf{FD}] \quad \frac{\sigma <: \sigma' \qquad \mathit{IdConv}(\sigma, \sigma')}{\sigma{*}/{?} <: \sigma'{*}/{?}}\,[\mathsf{Stream}]$$

$$\frac{}{\sigma{*} <: \texttt{object}}\,[\mathsf{SBox}] \quad \frac{}{\sigma{?} <: \sigma{*}}\,[\mathsf{SSub}] \quad \frac{}{\sigma <: \sigma{?}}\,[\mathsf{Sing}]$$

$$\frac{\overline{fd} <: \overline{fd'} \qquad \mathit{IdConv}(\overline{fd}, \overline{fd'})}{\texttt{struct}\{\overline{fd}\} <: \texttt{struct}\{\overline{fd'}\}}\,[\mathsf{Struct}] \quad \frac{}{\kappa <: \texttt{choice}\{\kappa;\ \overline{\kappa'}\}}\,[\mathsf{SubChoice}]$$

$$\frac{}{\texttt{choice}\{\overline{\kappa}\} <: \texttt{choice}\{\overline{\kappa}\,\overline{\kappa'}\}}\,[\mathsf{Choice}]$$

Most of these rules are straightforward. The rule [Stream] contains notation ($\sigma{*}/{?}$) that we use throughout this paper. It is uses to denote two instances of the rule, one where we select the left of the '/' in all cases (in this case $\sigma{*}$) and one where we select the right in all cases. It does *not* include cases where we individually select left and right alternatives. The rules [Stream] and [Struct] make use of a predicate IdConv, which relates two types τ and τ' if there is an identity conversion between them. Thus $\mathit{IdConv}(\texttt{Button}, \texttt{object})$ but not $\mathit{IdConv}(\texttt{int}, \texttt{object})$. In this short paper we shall not give its straightforward definition.

Generalized Member Access. As we have seen a key programming feature of Cω is generalized member access. Capturing this behaviour in the type system can be tricky, but we have adopted a rather elegant solution, whereby we define two auxiliary relations. The first, written $\tau.f : \tau'$, tells us that given a value of type τ accessing member f will return a value of type τ'. We define a similar relation for function member access, written $\tau.m(\overline{\tau'}) : \tau''$. Having generalized member access captured by a separate typing relation greatly simplifies the typing judgements for expressions. As generalized member access is a key feature of Cω, we shall give it in detail.

The definition of this relation over stream types is as follows.

$$\frac{\sigma.f : \sigma'}{\sigma{*}.f : \sigma'{*}} \quad \frac{\sigma.f : \sigma'{*}/{?}}{\sigma{*}.f : \sigma'{*}} \quad \frac{\sigma.m(\overline{\tau}) : \sigma'}{\sigma{*}.m(\overline{\tau}) : \sigma'{*}} \quad \frac{\sigma.m(\overline{\tau}) : \sigma'{*}/{?}}{\sigma{*}.m(\overline{\tau}) : \sigma'{*}} \quad \frac{\sigma.m(\overline{\tau}) : \texttt{void}}{\sigma{*}.m(\overline{\tau}) : \texttt{void}}$$

The first two rules map the member access over the stream elements, making sure that we do not create a nested stream type. The next two rules for function member access are similar. The last rule captures the intuition that mapping a `void`-valued method over a stream, forces the evaluation of the stream and does not return a value.

Before defining the rules for member access over anonymous structs, we need to define rules for member access over named field definitions. This is pretty straightforward and as follows.

$$\frac{}{\tau\, f.f : \tau} \quad \frac{\tau.m(\overline{\tau'}) : \tau''}{\tau\, f.m(\overline{\tau'}) : \tau''}$$

Now we consider the rules for generalized member access over anonymous structs. First we give the degenerate cases where only one component supports the member access.

$$\frac{\exists ! k \in \{1 \ldots n\}.\ fd_k.f\colon \tau_k}{\texttt{struct}\{fd_1\,;\ldots fd_n\,;\}.f\colon \tau_k} \qquad \frac{\exists ! k \in \{1 \ldots n\}.\ fd_k.m(\overline{\tau'})\colon \tau''}{\texttt{struct}\{fd_1\,;\ldots fd_n\,;\}.m(\overline{\tau})\colon \tau''}$$

The non-degenerate cases are then as follows.

$$\frac{\exists S \subseteq \{1 \ldots n\}.|S| \geq 2 \wedge p = |S| \wedge \forall k \in [1..p].fd_{S_k}.f\colon \tau_k}{\texttt{struct}\{fd_1\,;\ldots fd_n\,;\}.f\colon \texttt{struct}\{\tau_1;\ldots \tau_p;\}}$$

$$\frac{\exists S \subseteq \{1 \ldots n\}.|S| \geq 2 \wedge p = |S| \wedge \forall k \in [1..p].\ fd_{S_k}.m(\overline{\tau})\colon \tau'_k}{\texttt{struct}\{fd_1\,;\ldots fd_n\,;\}.m(\overline{\tau})\colon \texttt{struct}\{\tau'_1\,;\ldots \tau'_p\,;\}}$$

Thus a subset, S, of the components support the member, and we map the member access over these components in order. The overall return type is an anonymous struct of the component return types.

We now consider the rules for generalized member access over choice types. Again we consider these rules depending on how many components support the member access. First we give the simple case when *all* possible components support the member access.

$$\frac{\forall k \in \{1 \ldots n\}.\ \kappa_k.f\ :\ \tau}{\texttt{choice}\{\kappa_1\,;\ldots \kappa_n\,;\}.f\ :\ \tau} \qquad \frac{\forall k \in \{1 \ldots n\}.\ \kappa_k.m(\overline{\tau})\ :\ \tau'}{\texttt{choice}\{\kappa_1\,;\ldots \kappa_n\,;\}.m(\overline{\tau})\ :\ \tau'}$$

We also have the case when only one of the possible components supports the member access. These rules are as follows (we omit the nested cases).

$$\frac{\exists ! k \in \{1 \ldots n\}.\ \kappa_k.f\ :\ \sigma \qquad n > 1}{\texttt{choice}\{\kappa_1\,;\ldots \kappa_n\,;\}.f\ :\ \sigma\texttt{?}} \qquad \frac{\exists ! k \in \{1 \ldots n\}.\ \kappa_k.m(\overline{\tau})\ :\ \sigma \qquad n > 1}{\texttt{choice}\{\kappa_1\,;\ldots \kappa_n\,;\}.m(\overline{\tau})\ :\ \sigma\texttt{?}}$$

The reader will recall that the return type of this generalized member access involves a singleton stream type. Finally we give the cases where more than one of the possible components supports the member access.

$$\frac{\exists S \subseteq \{1 \ldots n\}.|S| \geq 2 \wedge p = |S| \wedge \forall k \in [1..p].\ \kappa_{S_k}.f\ :\ \kappa'_k}{\texttt{choice}\{\kappa_1\,;\ldots \kappa_n\,;\}.f\ :\ \texttt{choice}\{\kappa'_1\,;\ldots \kappa'_p\,;\}\texttt{?}}$$

$$\frac{\exists S \subseteq \{1 \ldots n\}.|S| \geq 2 \wedge p = |S| \wedge \forall k \in [1..p].\ \kappa_{S_k}.m(\overline{\tau})\ :\ \kappa'_k}{\texttt{choice}\{\kappa_1\,;\ldots \kappa_n\,;\}.m(\overline{\tau})\ :\ \texttt{choice}\{\kappa'_1\,;\ldots \kappa'_p\,;\}\texttt{?}}$$

Generalized member access over singleton streams is relatively straightforward; the only complication being again to ensure that no nested streams are generated.

$$\frac{\sigma.f\colon \sigma'}{\sigma\texttt{?}.f\colon \sigma'\texttt{?}} \qquad \frac{\sigma.f\colon \sigma'\texttt{*}/\texttt{?}}{\sigma\texttt{?}.f\colon \sigma'\texttt{*}/\texttt{?}} \qquad \frac{\sigma.m(\overline{\tau})\colon \sigma'}{\sigma\texttt{?}.m(\overline{\tau})\colon \sigma'\texttt{?}} \qquad \frac{\sigma.m(\overline{\tau})\colon \sigma'\texttt{*}/\texttt{?}}{\sigma\texttt{?}.m(\overline{\tau})\colon \sigma'\texttt{*}/\texttt{?}}$$

Finally we need to define rules for generalized member access over classes. Clearly these need to reflect the standard C$^\sharp$ semantics: function member access on classes searches the class hierarchy until a matching method is found. If we find a matching method $\tau' m(\overline{\tau''})$ in class c, we need to check the actual types of the arguments to the types expected by m. This behaviour is given by the following two rules.

$$\frac{\texttt{class}\ c\!:\!c'\texttt{\{}\tau\texttt{;}\overline{md}\texttt{\}} \quad \tau' m(\overline{\tau''}) \in \overline{md} \quad \overline{\tau} <: \overline{\tau''}}{c.m(\overline{\tau})\!:\tau'}$$

$$\frac{\texttt{class}\ c\!:\!c'\texttt{\{}\tau\texttt{;}\overline{md}\texttt{\}} \quad \tau' m(\overline{\tau''}) \notin \overline{md} \quad c'.m(\overline{\tau})\!:\tau'}{c.m(\overline{\tau})\!:\tau'}$$

Next we consider the rules for generalized field access. There is a small subtlety here concerning recursive class definitions; consider the following recursive class `List` of lists of integers: `class List { struct{ int head; List; } }`

Given an instance `xs` of type `List`, we do not want `xs.head` to recursively select all `head` fields in `xs`. However simply unfolding the content type and using the rules given earlier for generalized access over anonymous structs that is precisely what would happen!

There are a number of solutions, but in order to make the Cω type system as simple as possible, we follow e.g. Haskell and SML and break recursive cycles at nominal types. In our setting that means that we simply do not perform member lookup on nominal members of the content of nominal types. Using these refined rules, the result type of `xs.head` is `int`.

Formalizing this is trivial but time-consuming. We define another family of generalized member access judgements, written $\tau \bullet f\!:\tau'$, which is identical to the previous rules except they are not defined for nominal types. We elide the definitions here.

To define field access on nominal types, we first define formally the content type of a class, written $\mathit{content}(c)$ for some class c, as follows.

$$\frac{\texttt{class}\ c\texttt{:object\{}\tau\texttt{;}\overline{md}\texttt{\}}}{\mathit{content}(c) = \tau} \qquad \frac{\texttt{class}\ c : c'\texttt{\{}\tau\texttt{;}\overline{md}\texttt{\}} \quad \mathit{content}(c') = \tau'}{\mathit{content}(c) = \texttt{struct\{}\tau'\texttt{;}\tau\texttt{;\}}}$$

The rule for generalized member access over classes simply searches for the member f on the content type of class c, and is given by the following rule.

$$\frac{\mathit{content}(c) = \tau \quad \tau \bullet f : \tau'}{c.f\!:\tau'}$$

Generalized Index Access. As we mentioned earlier, elements of anonymous structs can be accessed by position. This is captured by the following rule.

$$\frac{\mathit{type}(\mathit{fd}_i) = \tau_i}{\texttt{struct \{}\mathit{fd}_1\texttt{;}\ldots\mathit{fd}_n\texttt{;\}[}i\texttt{]}\!:\tau_i}$$

As the reader might have expected, this index access is generalized over the other types; the rather routine details are omitted.

Typing Judgements. We are now able to define typing judgements for FCω. We define three relations corresponding to the three syntactic categories of expressions, promotable expressions and statements. For all three judgements we write Γ to mean a partial function from program identifiers to types. The judgements for expressions and promotable expressions are written $\Gamma \vdash e\colon\tau$ and $\Gamma \vdash pe\colon\tau$, respectively. These are given in Fig. 1.

Most of these rules are routine; we shall discuss a few of the more interesting details here. In the rule [TStruct], we have made use of a typing judgement for a binding expression. This is defined as follows:

$$\frac{\Gamma \vdash e\colon\tau}{\Gamma \vdash f = e\colon\tau\, f}$$

The compactness of the rules [TField], [TIndex] and [TMeth] shows the elegance of having captured generalized member access with auxiliary relations.

The rules [TAAExp1] and [TAAExp2] ensure that the return type of an apply-to-all expression is not nested. The rule [TAAExp3] ensures the appropriate mixed flattening of streams. The rule [TAAExp4] captures the intuition that applying a `void`-typed expression to a stream forces the evaluation of that stream and hence the overall type is also `void`.

The typing judgement for FCω statements is written $\Gamma;\tau \vdash s$ and is intended to mean that a statement s is well-typed in the typing environment Γ. If it returns a value (either via a normal `return` or a `yield return`) then that value is of type τ.

The rules [TForeach1] and [TForeach2] reflect the fact that the type of the stream elements can be cast to the type of the bound variable. This can be either via an upcast ([TForeach1]) or a downcast ([TForeach2]) (again this matches C$^\sharp$ 2.0).

3.2 An Inner Calculus: ICω

Rather than consider further our featherweight calculus FCω, we shall in fact define another core calculus for Cω. This inner calculus, called ICω, is intended to be similar but lower-level than FCω; it can be thought of as the internal language of a compiler.

The chief simplification in ICω is that its type system does *not* support generalized member access. The intention is that we compile out generalized member access when translating FCω programs into ICω programs. We give some details of this compilation in §3.4. Apart from a simplified type system, we can define quite simply an operational semantics for ICω; this is given in §3.3.

The grammar of ICω is then a simple varianr of the grammar for FCω. Some extra expression and statement forms are added (which reflects the lower-level nature of ICω) and likewise a couple are removed from the grammar as they are redundant. We do not expect these new syntactic forms to be made available to the Cω programmer (although they could be). The extensions are as follows:

Expression

$e ::= \ldots$		
$\mid$	`new` τ`(`$\overline{s}$`)`	Closure creation
$\mid$	`new (`κ`,`e`)`	Choice creation
$\mid$	e`.Content`	Class content
$\mid$	e `at` κ	Choice content

Promotable expression

$pe ::= \ldots$		
$\mid$	τ`({`$\overline{s}$`})`	Block expression

Statement

$s ::= \ldots$		
$\mid$	`yield return (`τ`,`e`);`	Typed yield

$\boxed{\Gamma \vdash e\colon \tau \text{ and } \Gamma \vdash pe\colon \tau}$

$$\frac{}{\Gamma \vdash i\colon \mathtt{int}}\,[\text{TInt}] \quad \frac{}{\Gamma \vdash b\colon \mathtt{bool}}\,[\text{TBool}] \quad \frac{}{\Gamma, x\colon \tau \vdash x\colon \tau}\,[\text{TId}] \quad \frac{}{\Gamma \vdash \mathtt{null}\colon \mathtt{null}}\,[\text{TNull}]$$

$$\frac{\Gamma \vdash e\colon \tau' \quad (\tau' <: \tau) \vee (\tau <: \tau')}{\Gamma \vdash (\tau)\, e\colon \tau}\,[\text{TSub}] \qquad \frac{\Gamma \vdash e\colon \tau' \quad (\tau' <: \tau) \vee (\tau <: \tau')}{\Gamma \vdash e \;\mathtt{is}\; \tau\colon \mathtt{bool}}\,[\text{TIs}]$$

$$\frac{\Gamma \vdash e\colon \mathtt{choice\{}\overline{\kappa'}\;\kappa;\overline{\kappa''}\mathtt{\}}}{\Gamma \vdash e \;\mathtt{was}\; \kappa\colon \mathtt{bool}}\,[\text{TWas}] \qquad \frac{\Gamma \vdash \overline{be}\colon \overline{fd}}{\Gamma \vdash \mathtt{new}\;\mathtt{\{}\overline{be}\mathtt{\}}\colon \mathtt{struct\{}\overline{fd}\mathtt{\}}}\,[\text{TStruct}]$$

$$\frac{\Gamma \vdash e\colon \tau \quad \tau <: \mathit{content}(c)}{\Gamma \vdash \mathtt{new}\; c\mathtt{(}e\mathtt{)}\colon c}\,[\text{TNew}] \qquad \frac{\Gamma \vdash e\colon \tau \quad \tau.f : \tau'}{\Gamma \vdash e\mathtt{.}f\colon \tau'}\,[\text{TField}]$$

$$\frac{\Gamma \vdash e\colon \tau \quad \tau[i] : \tau'}{\Gamma \vdash e\mathtt{[}i\mathtt{]}\colon \tau'}\,[\text{TIndex}] \qquad \frac{\Gamma \vdash x\colon \tau \quad \Gamma \vdash e\colon \tau' \quad \tau' <: \tau}{\Gamma \vdash x\mathtt{=}e\colon \tau}\,[\text{TAss}]$$

$$\frac{\Gamma \vdash e\colon \tau \quad \Gamma \vdash \overline{e'}\colon \overline{\tau'} \quad \tau.m(\overline{\tau'})\colon \tau''}{\Gamma \vdash e\mathtt{.}m\mathtt{(}\overline{e'}\mathtt{)}\colon \tau''}\,[\text{TMeth}] \quad \frac{\Gamma \vdash e\colon \sigma{*}/? \quad \Gamma, \mathtt{it} : \sigma \vdash e'\colon \sigma'}{\Gamma \vdash e\mathtt{.\{}e'\mathtt{\}}\colon \sigma'{*}/?}\,[\text{TAAExp1}]$$

$$\frac{\Gamma \vdash e\colon \sigma{*}/? \quad \Gamma, \mathtt{it} : \sigma \vdash e'\colon \sigma'{*}/?}{\Gamma \vdash e\mathtt{.\{}e'\mathtt{\}}\colon \sigma'{*}/?}\,[\text{TAAExp2}] \quad \frac{\Gamma \vdash e\colon \sigma{*}/? \quad \Gamma, \mathtt{it} : \sigma \vdash e'\colon \sigma'?/{*}}{\Gamma \vdash e\mathtt{.\{}e'\mathtt{\}}\colon \sigma'{*}}\,[\text{TAAExp3}]$$

$$\frac{\Gamma \vdash e\colon \sigma{*}/? \quad \Gamma, \mathtt{it} : \sigma \vdash e'\colon \mathtt{void}}{\Gamma \vdash e\mathtt{.\{}e'\mathtt{\}}\colon \mathtt{void}}\,[\text{TAAExp4}]$$

$\boxed{\Gamma; \tau \vdash s}$

$$\frac{}{\Gamma; \tau \vdash \mathtt{;}}\,[\text{TSkip}] \quad \frac{\Gamma; \tau \vdash \overline{s}}{\Gamma; \tau \vdash \mathtt{\{}\overline{s}\mathtt{\}}}\,[\text{TNest}] \quad \frac{\Gamma \vdash pe\colon \tau}{\Gamma; \tau' \vdash pe\mathtt{;}}\,[\text{TProm}] \quad \frac{}{\Gamma; \mathtt{void} \vdash \mathtt{return;}}\,[\text{TRetV}]$$

$$\frac{\Gamma \vdash e\colon \mathtt{bool} \quad \Gamma; \tau \vdash s}{\Gamma; \tau \vdash \mathtt{while}\;\mathtt{(}e\mathtt{)}\; s}\,[\text{TWhile}] \qquad \frac{\Gamma \vdash e\colon \mathtt{bool} \quad \Gamma; \tau \vdash s_1 \quad \Gamma; \tau \vdash s_2}{\Gamma; \tau \vdash \mathtt{if}\;\mathtt{(}e\mathtt{)}\; s_1 \;\mathtt{else}\; s_2}\,[\text{TIf}]$$

$$\frac{\Gamma \vdash e\colon \tau' \quad \tau' <: \tau}{\Gamma; \tau \vdash \mathtt{return}\; e\mathtt{;}}\,[\text{TRet}] \qquad \frac{}{\Gamma; \sigma{*} \vdash \mathtt{yield\;break;}}\,[\text{TYieldB}]$$

$$\frac{\Gamma \vdash e\colon \sigma' \quad \sigma' <: \sigma}{\Gamma; \sigma{*} \vdash \mathtt{yield\;return}\; e\mathtt{;}}\,[\text{TYield1}] \quad \frac{\Gamma \vdash e\colon \sigma'{*}/? \quad \sigma' <: \sigma'' \quad \Gamma, x\colon \sigma''; \tau \vdash s}{\Gamma; \tau \vdash \mathtt{foreach}\;\mathtt{(}\sigma''\; x \;\mathtt{in}\; e\mathtt{)}s}\,[\text{TForeach1}]$$

$$\frac{\Gamma \vdash e\colon \sigma{*} \quad \sigma{*} <: \sigma'{*}}{\Gamma; \sigma'{*} \vdash \mathtt{yield\;return}\; e\mathtt{;}}\,[\text{TYield2}] \quad \frac{\Gamma \vdash e\colon \sigma'{*}/? \quad \sigma'' <: \sigma' \quad \Gamma, x\colon \sigma''; \tau \vdash s}{\Gamma; \tau \vdash \mathtt{foreach}\;\mathtt{(}\sigma''\; x \;\mathtt{in}\; e\mathtt{)}s}\,[\text{TForeach2}]$$

Fig. 1. Typing judgements for FCω expressions, promotable expressions and statements

Thus ICω includes expressions to create closure and choice elements. We include an operator e`.Content` to extract the content element from an object e. Given an element e of a choice type, we add an operation e `at` κ to extract its κ-valued content. (If it is of another type, this will raise an exception.) We add (typed) block expressions to ICω, and in addition we provide a typed `yield` statement.

The two syntactic forms that we removed from the grammar of FCω are: (1) We remove field accesses $e.f$ completely; they are replaced by positional access, i.e. $e[i]$; and (2) We remove the untyped `yield` statement; all `yields` in ICω are explicitly typed.

We can define typing judgements for ICω expressions and statements, which are written $\Gamma \rhd e\colon \tau$ and $\Gamma; \tau \rhd s$, respectively. Most of these rules are identical to those for FCω; we shall just give the rules for the new syntactic forms. The rules for creating closure and choice elements are as follows:

$$\frac{\Gamma; \sigma{*}/? \rhd \overline{s}}{\Gamma \rhd \texttt{new}\ \sigma{*}/?(\overline{s})\colon \sigma{*}/?} \qquad \frac{\Gamma \rhd e\colon \kappa' \quad \kappa' <: \kappa}{\Gamma \rhd \texttt{new}\ (\kappa, e)\colon \texttt{choice\{}\kappa\texttt{;\}}}$$

The typing rules for extracting the content of content class and choice elements are as follows:

$$\frac{\Gamma \rhd e\colon c}{\Gamma \rhd e\texttt{.Content}\colon \mathit{content}(c)} \qquad \frac{\Gamma \rhd e\colon \texttt{choice\{}\kappa\texttt{;}\overline{\kappa'}\texttt{\}}}{\Gamma \rhd e\ \texttt{at}\ \kappa\colon \kappa}$$

The typing rule for block expressions and `yield` statements are as follows:

$$\frac{\Gamma; \tau \rhd \overline{s} \quad \tau \neq \texttt{void}}{\Gamma \rhd \tau\texttt{(\{}\overline{s}\texttt{\})}\colon \tau} \qquad \frac{}{\Gamma; \sigma{*}/? \vdash \texttt{yield break;}}$$

$$\frac{\Gamma \rhd e\colon \sigma' \quad \sigma' <: \sigma}{\Gamma; \sigma{*}/? \rhd \texttt{yield return}\ (\sigma', e)\texttt{;}} \qquad \frac{\Gamma \rhd e\colon \sigma{*}/? \quad \sigma{*}/? <: \tau \quad \tau \neq \texttt{object}}{\Gamma; \tau \rhd \texttt{yield return}\ (\sigma{*}/?, e)\texttt{;}}$$

3.3 Operational Semantics for ICω

In this section we formalize the dynamics of ICω by defining an operational semantics. We follow FJ [18] and MJ [5] and give this in the form of a small-step reduction relation, although a big-step evaluation relation can easily be defined. Hence we use evaluation contexts to encode the evaluation strategy in the now familiar way [11]—the definition of ICω evaluation contexts is routine and omitted. First we define the value forms of ICω expressions and statements (where bv is the value form of a binding expression):

Expression values		**Statement values**	
$v ::= b \mid i \mid$ `null` $\mid$ `void`	Basic values	$sv ::=$ `;`	Skip
$\mid\ r$	Reference	$\mid$ `return` v`;`	Return value
$\mid$ `new {`$\overline{bv}$`}`	Struct value	$\mid$ `return;`	
$\mid$ `new (`κ`,`v`)`	Choice value	$\mid$ `yield return (`τ`,`v`);`	Typed yield value
		$\mid$ `yield break;`	End of stream value

Evaluation of ICω expressions and statements takes place in the context of a state, which is a pair (H, R), where H is a heap and R is a stack frame. A heap is represented as a finite partial map from references r to runtime objects, and a stack frame is a finite partial map from variable identifiers to values. A runtime object, as for C$^\sharp$, is a pair (τ, cn) where τ is a type and cn is a canonical, which is either a value or a closure. A closure is the runtime representation of a stream and is written as a pair $(R, \overline{s})^\alpha$ where R is a stack frame and $\overline{s}$ is a statement sequence. The superscript flag α indicates whether the closure is fresh or a clone. We will explain this distinction later. In what follows we assume that expressions and statements are well-typed.

In Fig. 2 we define the evaluation relation for ICω expressions, written $S, e \to S', e'$, which means that given a state S, expression e reduces by one or possibly more steps to e' and a (possibly updated) state S'. (We use an auxiliary function $value$ defined as follows: $value(f = v) \stackrel{\text{def}}{=} v$, $value(v) \stackrel{\text{def}}{=} v$.) These rules are routine.

As is usual we have a number of cases that lead to a predicable error state, e.g. following a dereference of a `null` object. These errors in ICω are $CastX$, $ChoiceX$, $NullX$ and $NullableX$. We say that a pair S, e is *terminal* if e is one of these errors, or it is a value.

The evaluation relation for ICω promotable expressions is written $S, pe \to S', pe'$ and is also given in Fig. 2. The rules for method invocation deserve some explanation: they are differentiated according to whether the method is `void`-returning. If it is not then the method body is unfolded, and executed until it is of the form `return` v`;` where v is a value. This value is then the result of the method invocation. If the method is `void`-valued, then we unfold the method body and execute it until it is of the form `return;`. The result is the special value `void`.

The evaluation relation for statements is written $S, s \to S', s'$ and in Fig. 3 we give just some of the interesting cases, which are those dealing with `foreach` loops. As we have mentioned, Cω streams are aligned with C$^\sharp$ 2.0 iterators: there the `foreach` loop is actually syntactic sugar: first of all an `IEnumerator<T>` is obtained from the iterator block (which should be of type `IEnumerable<T>`) using the `GetEnumerator` method. This is walked over using `MoveNext` and `Current` members. Semantically important is that `GetEnumerator` actually copies the enumerable object. In our semantics we faithfully encode this by tagging closures, and creating clones as appropriate. Thus whilst iterating over a stream we update the reference in place (rules [FVC], [FSC] and [FNC]), but every `foreach` creates its own copy from a fresh original (rules [FVF], [FSF] and [FNF]). In rule [FBr] we write α to range over both clone and fresh.

Rules [FSF] and [FSC] embody the flattening of streams. To evaluate a `foreach` loop we first evaluate the stream until it `yield`s a value. If that value is itself a stream, then we should first execute the `foreach` loop on this stream.

3.4 Compiling FCω to ICω

In this section we give some details of the compilation of FCω into ICω. Much of this compilation is routine, so in the interests of space we shall concentrate only on the most interesting aspect: generalized member access.

We employ a ‘‘coercion” technique, in that we translate the *implicit* generalized member access of FCω into an *explicit* ICω code fragment. This can be expressed as an inductively defined relation, written $|\tau.f\colon \tau'| \rightsquigarrow g$ and $|\tau.m(\overline{\tau'})\colon \tau''| \rightsquigarrow g$ for member and function member access respectively. A judgement $|\tau.f\colon \tau'| \rightsquigarrow g$ is intended to

$\boxed{\text{Expressions}}$

$$\frac{}{(H,R), x \to (H,R), R(x)} \qquad \frac{H(r) = (\tau', cn) \quad \tau' <: \tau}{(H,R), \texttt{(}\tau\texttt{)}\, r \to (H,R), r}$$

$$\frac{H(r) = (\tau', cn) \quad \tau' \not<: \tau}{(H,R), \texttt{(}\tau\texttt{)}\, r \to (H,R), \mathit{CastX}} \qquad \frac{H(r) = (\tau', cn) \quad \tau' <: \tau}{(H,R), r\ \texttt{is}\ \tau \to (H,R), \texttt{true}}$$

$$\frac{H(r) = (\tau', cn) \quad \tau' \not<: \tau}{(H,R), r\ \texttt{is}\ \tau \to (H,R), \texttt{false}} \qquad \frac{}{S, \texttt{new (}\kappa, v\texttt{) was}\ \kappa \to S, \texttt{true}}$$

$$\frac{\kappa \neq \kappa'}{S, \texttt{new (}\kappa, v\texttt{) was}\ \kappa' \to S, \texttt{false}} \quad \frac{r \notin \mathit{dom}(H)}{(H,R), \texttt{new}\ c\texttt{(}v\texttt{)} \to (H[r \mapsto (c, v)], R), r}$$

$$\frac{r \notin \mathit{dom}(H)}{(H,R), \texttt{new}\ \sigma\texttt{*/?(}\overline{s}\texttt{)} \to (H[r \mapsto (\sigma\texttt{*/?}, (R, \overline{s})^{\mathsf{fresh}})], R), r}$$

$$\frac{H(r) = (c, cn)}{(H,R), r\texttt{.content} \to (H,R), cn} \qquad \frac{}{S, \texttt{null.content} \to S, \mathit{NullX}}$$

$$\frac{0 \le i \le n}{\begin{array}{l}S, \texttt{new \{}bv_0, .., bv_n\texttt{\}[}i\texttt{]} \to \\ S, \mathit{value}(bv_i)\end{array}} \qquad \frac{}{S, \texttt{new (}\kappa, v\texttt{) at}\ \kappa \to S, v} \qquad \frac{\kappa \neq \kappa'}{\begin{array}{l}S, \texttt{new (}\kappa, v\texttt{) at}\ \kappa' \to \\ S, \mathit{ChoiceX}\end{array}}$$

$\boxed{\text{Promotable expressions}}$

$$\frac{}{(H,R), x\ \texttt{=}\ v \to (H, R[x \mapsto v]), v} \qquad \frac{(H,R), \overline{s} \to^{*} (H', R'), \texttt{return}\ v\texttt{;}\ \overline{s'}}{(H,R), \tau\texttt{(\{}\overline{s}\texttt{\})} \to (H', R'), v}$$

$$\frac{}{S, \texttt{null.}m\texttt{(}\overline{v}\texttt{)} \to S, \mathit{NullX}} \quad \frac{\begin{array}{c}H(r) = (c, \_) \quad \mathit{method}(m, c) = \tau'(\overline{\tau}\ \overline{x})\{\overline{s}\} \quad \tau' \neq \texttt{void} \\ (H, [\,]), \texttt{\{}c\ \texttt{this = }r\texttt{;}\overline{\tau}\ \overline{x}\ \texttt{=}\ \overline{v}\texttt{;}\overline{s}\texttt{\}} \to^{*} (H', R'), \texttt{return}\ v'\texttt{;}\overline{s'}\end{array}}{(H,R), r\texttt{.}m\texttt{(}\overline{v}\texttt{)} \to (H', R), v'}$$

$$\frac{\begin{array}{c}H(r) = (c, \_) \quad \mathit{method}(m, c) = \texttt{void}\ (\overline{\tau}\ \overline{x})\{\overline{s}\} \\ (H, [\,]), \texttt{\{}c\ \texttt{this = }r\texttt{;}\overline{\tau}\ \overline{x}\ \texttt{=}\ \overline{v}\texttt{;}\overline{s}\texttt{\}} \to^{*} (H', R'), \texttt{return ;}\overline{s'}\end{array}}{(H,R), r\texttt{.}m\texttt{(}\overline{v}\texttt{)} \to (H', R), \texttt{void}}$$

Fig. 2. Evaluation rules for ICω expressions and promotable expressions

mean that if invoking a member f on an element of type τ returns an element of type τ', then g is the ICω coercion that encodes the explicit access of the appropriate member. In Fig. 4 we give some details of the compilation of generalized member access (GMA) for members, i.e. the $|\tau.f\!:\tau'| \rightsquigarrow g$ relation. (The version for function members

$$\frac{}{S, \texttt{foreach (}\sigma\ x\ \texttt{in null)}s \to S, \texttt{;}}\ \text{[FNull]}$$

$$\frac{H(r) = (\tau', (R', \overline{s'})^{\alpha}) \quad (H, R'), \overline{s'} \to^* (H', R''), \texttt{yield break ;}\overline{s''}}{(H, R), \texttt{foreach (}\sigma\ x\ \texttt{in}\ r\texttt{)}\ s \to (H', R), \texttt{;}}\ \text{[FBr]}$$

$$\frac{\begin{array}{ll} H(r) = (\tau', (R', \overline{s'})^{\mathsf{fresh}}) & r' \notin dom(H') \\ (H, R'), \overline{s'} \to^* (H', R''), \texttt{yield return (}\sigma', v\texttt{)}\,\texttt{;}\overline{s''} & v \neq \texttt{null} \end{array}}{\begin{array}{l} (H, R), \texttt{foreach (}\sigma\ x\ \texttt{in}\ r\texttt{)}\ s \to \\ (H'[r' \mapsto (\tau', (R'', \overline{s''})^{\mathsf{clone}})], R), \texttt{\{\{}\sigma\ x\ \texttt{=}\ v; s\texttt{\}}\ \texttt{foreach (}\sigma\ x\ \texttt{in}\ r'\texttt{)}\ s\texttt{\}} \end{array}}\ \text{[FVF]}$$

$$\frac{\begin{array}{l} H(r) = (\tau', (R', \overline{s'})^{\mathsf{clone}}) \\ (H, R'), \overline{s'} \to^* (H', R''), \texttt{yield return (}\sigma', v\texttt{)}\,\texttt{;}\overline{s''} \quad v \neq \texttt{null} \end{array}}{\begin{array}{l} (H, R), \texttt{foreach (}\sigma\ x\ \texttt{in}\ r\texttt{)}\ s \to \\ (H'[r \mapsto (\tau', (R'', \overline{s''})^{\mathsf{clone}})], R), \texttt{\{\{}\sigma\ x\ \texttt{=}\ v; s\texttt{\}}\ \texttt{foreach (}\sigma\ x\ \texttt{in}\ r\texttt{)}\ s\texttt{\}} \end{array}}\ \text{[FVC]}$$

$$\frac{\begin{array}{ll} H(r) = (\tau', (R', \overline{s'})^{\mathsf{fresh}}) & r' \notin dom(H') \\ (H, R'), \overline{s'} \to^* (H', R''), \texttt{yield return (}\sigma'\texttt{*}, v\texttt{)}\,\texttt{;}\overline{s''} & v \neq \texttt{null} \end{array}}{\begin{array}{l} (H, R), \texttt{foreach (}\sigma\ x\ \texttt{in}\ r\texttt{)}\ s \to \\ (H'[r' \mapsto (\tau', (R'', \overline{s''})^{\mathsf{clone}})], R), \texttt{\{foreach (}\sigma\ x\ \texttt{in}\ v\texttt{)}\ s\ \texttt{foreach (}\sigma\ x\ \texttt{in}\ r'\texttt{)}\ s\texttt{\}} \end{array}}\ \text{[FSF]}$$

$$\frac{\begin{array}{l} H(r) = (\tau', (R', \overline{s'})^{\mathsf{clone}}) \\ (H, R'), \overline{s'} \to^* (H', R''), \texttt{yield return (}\sigma'\texttt{*}, v\texttt{)}\,\texttt{;}\overline{s''} \quad v \neq \texttt{null} \end{array}}{\begin{array}{l} (H, R), \texttt{foreach (}\sigma\ x\ \texttt{in}\ r\texttt{)}\ s \to \\ (H'[r \mapsto (\tau', (R'', \overline{s''})^{\mathsf{clone}})], R), \texttt{\{foreach (}\sigma\ x\ \texttt{in}\ v\texttt{)}\ s\ \texttt{foreach (}\sigma\ x\ \texttt{in}\ r\texttt{)}\ s\texttt{\}} \end{array}}\ \text{[FSC]}$$

$$\frac{\begin{array}{ll} H(r) = (\tau', (R', \overline{s'})^{\mathsf{fresh}}) & r' \notin dom(H') \\ (H, R'), \overline{s'} \to^* (H', R''), \texttt{yield return (}\tau, \texttt{null)}\,\texttt{;}\overline{s''} & \end{array}}{\begin{array}{l} (H, R), \texttt{foreach (}\sigma\ x\ \texttt{in}\ r\texttt{)}\ s \to \\ (H'[r' \mapsto (\tau', (R'', \overline{s''})^{\mathsf{clone}})], R), \texttt{foreach (}\sigma\ x\ \texttt{in}\ r'\texttt{)}\ s \end{array}}\ \text{[FNF]}$$

$$\frac{\begin{array}{l} H(r) = (\tau', (R', \overline{s'})^{\mathsf{clone}}) \\ (H, R'), \overline{s'} \to^* (H', R''), \texttt{yield return (}\tau, \texttt{null)}\,\texttt{;}\overline{s''} \end{array}}{(H, R), \texttt{foreach (}\sigma\ x\ \texttt{in}\ r\texttt{)}\ s \to (H'[r \mapsto (\tau', (R'', \overline{s''})^{\mathsf{clone}})], R), \texttt{foreach (}\sigma\ x\ \texttt{in}\ r\texttt{)}\ s}\ \text{[FNC]}$$

Fig. 3. Evaluation rules for ICω `foreach` loops

(methods) is similar and omitted.) In the definition we have employed a function-like syntax for coercions, although they are really contexts, and we have dropped the types from various block expressions. We have used the shorthand `yield return`$'$`(`τ`,e);` to mean the statement sequence `yield return(`τ`,e);yield break;` and have also used two functions: `Value` that returns the element of a singleton stream or raises an exception if it empty, and `HasValue` that returns a boolean depending on whether the singleton stream has an element or not. These can be coded directly and their definitions are omitted.

$\boxed{\text{Compiling GMA over streams}}$

$$\frac{|\sigma.f\!:\sigma'| \leadsto g}{|\sigma\texttt{*}.f\!:\sigma'\texttt{*}| \leadsto z \mapsto z\texttt{.\{}g(\texttt{it})\texttt{\}}}$$

$\boxed{\text{Compiling GMA over anonymous structs}}$

$$\frac{\exists S \subseteq \{1\ldots n\}.|S| \geq 2. \wedge p = |S| \wedge \forall k \in [1..p].\ |fd_{S_k}.f : \tau_k| \leadsto g_k}{|\texttt{struct\{}fd_1\texttt{;}\ldots fd_n\texttt{;\}}.f\!:\texttt{struct}\{\tau_1;\ldots\tau_p;\}| \leadsto z \mapsto \texttt{new\{}g_1(z\texttt{[1]}),\ldots,g_p(z\texttt{[}p\texttt{]})\texttt{\}}}$$

$$\frac{\exists!k \in \{1\ldots n\}.\ |fd_k.f : \tau_k| \leadsto g}{|\texttt{struct\{}fd_1\texttt{;}\ldots fd_n\texttt{;\}}.f\!:\tau_k| \leadsto z \mapsto g(z\texttt{[}k\texttt{]})}$$

$\boxed{\text{Compiling GMA over choice types}}$

$$\frac{\exists S \subseteq \{1\ldots n\}.|S| \geq 2 \wedge p = |S| \wedge \forall k \in [1..p].\ |\kappa_{S_k}.f : \kappa'_k| \leadsto g_k}{\begin{array}{l}|\texttt{choice}\{\kappa_1\texttt{;}\ldots\kappa_n\texttt{;}\}.f : \texttt{choice}\{\kappa'_1\texttt{;}\ldots\kappa'_p\texttt{;}\}\texttt{?}| \\ \leadsto z \mapsto \\ \quad \texttt{(\{if(}z\ \texttt{was}\ \kappa_{S_1}\texttt{)} \\ \quad\ \texttt{\{return new choice\{}\kappa'_1\texttt{;}\ldots\kappa'_p\texttt{;\}?(yield return}'\texttt{(}\kappa_{S_1}\texttt{,new(}\kappa_{S_1}\texttt{,}g_1(z\ \texttt{at}\ \kappa_{S_1})\texttt{));)\texttt{;\}} \\ \quad \cdots\texttt{if(}z\ \texttt{was}\ \kappa_{S_p}\texttt{)} \\ \quad\ \texttt{\{return new choice\{}\kappa'_1\texttt{;}\ldots\kappa'_p\texttt{;\}?(yield return}'\texttt{(}\kappa_{S_p}\texttt{,new(}\kappa_{S_p}\texttt{,}g_p(z\ \texttt{at}\ \kappa_{S_p})\texttt{));)\texttt{;\}} \\ \quad \texttt{else return null;\})}\end{array}}$$

$$\frac{|\kappa_k.f : \tau| \leadsto g_k \quad \forall k.1 \leq k \leq n}{\begin{array}{r}|\texttt{choice}\{\kappa_1\texttt{;}\ldots\kappa_n\texttt{;}\}.f : \tau| \leadsto z \mapsto \texttt{(\{ if(}z\ \texttt{was}\ \kappa_1\texttt{) return}\ g_1(z\ \texttt{at}\ \kappa_1)\texttt{;}\cdots \\ \texttt{if(}z\ \texttt{was}\ \kappa_n\texttt{) return}\ g_n(z\ \texttt{at}\ \kappa_n)\texttt{;\})}\end{array}}$$

$$\frac{\exists!k \in \{1\ldots n\}.\ |\kappa_k.f : \sigma| \leadsto g \quad n > 1}{\begin{array}{l}|\texttt{choice}\{\kappa_1\texttt{;}\ldots\kappa_n\texttt{;}\}.f : \sigma\texttt{?}| \\ \leadsto z \mapsto \texttt{(\{if(}z\ \texttt{was}\ \kappa_k\texttt{) return new}\ \sigma\texttt{?(yield return}'\texttt{(}\sigma\texttt{,}g(z\ \texttt{at}\ \kappa_k)\texttt{));)\texttt{;} \\ \qquad\quad \texttt{else return null;\})}\end{array}}$$

$\boxed{\text{Compiling GMA over singleton streams}}$

$$\frac{|\sigma.f\!:\sigma'| \leadsto g}{\begin{array}{l}|\sigma\texttt{?}.f\!:\sigma'\texttt{?}| \leadsto z \mapsto \texttt{(\{if (HasValue(}z\texttt{)) return new}\ \sigma'\texttt{?(yield return}'\texttt{(}\sigma'\texttt{,}g(\texttt{Value(}z\texttt{)}))\texttt{;)\texttt{;} \\ \qquad\qquad\qquad\quad \texttt{else return null;\})}\end{array}}$$

$$\frac{|\sigma.f\!:\sigma'\texttt{*/?}| \leadsto g}{\begin{array}{r}|\sigma\texttt{?}.f\!:\sigma'\texttt{*/?}| \leadsto z \mapsto \texttt{(\{if (HasValue(}z\texttt{)) return}\ g(\texttt{Value(}z\texttt{)})\texttt{;} \\ \texttt{else return null;\})}\end{array}}$$

Fig. 4. Compilation of Generalized Member Access

For example, we can compile an instance of member access in FCω, $e.f$, as follows: we first compile the expression e into ICω, yielding e', and also generate a coercion, g, corresponding to the member access. The result of the compilation of $e.f$ is then simply $g(e')$. We write the compilation of, e.g. an expression, e, as $|\Gamma \vdash e\colon \tau| \rightsquigarrow e'$.

Incoherence by Design. Java and C♯ are by design incoherent [7]. Both languages use a notion of ''best'' conversion when there is more than one conversion between two types. If there does not exist a best conversion, a compile-time error is generated. In compiling FCω to ICω we use this notion of a best conversion when dealing with rules that use subtyping. We do not formalize this notion of ''best'' here; both the Java and C♯ language specifications give precise details. The new types in Cω do not complicate this notion greatly: For example, there are two conversions between `int` and `object`: one using the rule [Box], the other using the rules [SubChoice] and [Box] along with [Trans] (i.e. `int` <: `choice{int;string;}` <: `object`). It is clear that the first conversion is better. The other critical pairs are similarly easy to resolve.

3.5 Properties of FCω and ICω

In this section we briefly mention some properties of FCω and ICω and the compilation. We do not give any details of the proofs, as they are standard and follow analogous theorems for Java [18, 5]; details will appear in a forthcoming technical report.

Our main result is that ICω is type-sound, which is captured by the following properties. (We use generalized judgements, e.g. $\Gamma \rhd (S, e)\colon \tau$ to mean that the expression e is well-typed and also that the state S is well-formed with respect to Γ, in the familiar way. As is usual [18] we also need to add ''stupid'' typing rules for the formal proof.)

Theorem 1 (Type soundness for ICω**).**

1. *If* $\Gamma \rhd (S, e)\colon \tau$ *and* $(S, e) \to (S', e')$ *then* $\exists \tau'$ *such that* $\Gamma \rhd (S', e')\colon \tau'$ *and* $\tau' <: \tau$.
2. *If* $\Gamma; \tau \rhd s$ *and* $(S, s) \to (S', s')$ *then* $\exists \tau'$ *such that* $\Gamma; \tau' \rhd (S', s')$ *and* $\tau' <: \tau$.
3. *If* $\Gamma \rhd (S, e)\colon \tau$ *then either* (S, e) *is terminal or* $\exists S', e'$ *such that* $(S, e) \to (S', e')$.
4. *If* $\Gamma; \tau \rhd (S, s)$ *then either* (S, s) *is terminal or* $\exists S', s'$ *such that* $(S, s) \to (S', s')$.

We can also prove that our compilation of FCω to ICω is type-preserving, i.e. if an FCω expression e in environment Γ has type τ , then there is a compilation of e resulting in an ICω expression e', such that e' in Γ also has type τ.

Theorem 2 (Type preservation of compilation).

1. *If* $\Gamma \vdash e\colon \tau$ *then* $\exists e'$ *such that* $|\Gamma \vdash e\colon \tau| \rightsquigarrow e'$ *and* $\Gamma \rhd e'\colon \tau$.
2. *If* $\Gamma; \tau \vdash s$ *then* $\exists s'$ *such that* $|\Gamma; \tau \vdash s| \rightsquigarrow s'$ *and* $\Gamma; \tau \rhd s'$.

4 Related Work

Numerous languages have been proposed for manipulating relational and semi-structured data. For reasons of space we focus here only on those for semi-structured data (some of the languages for relational data were cited in §1).

A number of special-purpose functional languages [15, 4, 10] have been proposed for processing XML values. This stands in contrast to our approach, which aims at extending an existing widely-used object-oriented programming language.

The languages most similar to Cω are XJ [14] and Xtatic [13]. XJ adds XML and XPath as a first-class construct to Java, and uses logical XML classes to represent XSDs. In this way XJ allows compile time checking of XML fragments; however since the impedance mismatch between XML and objects is quite large, it does not deal with a mix of data from the the object and the XML world. One consequence is, for example, that XPath queries are restricted to work on XML data only.

Xtatic extends C♯ with a separate category of regular expression types [16]. Subtyping is structural. While this gives a lot of flexibility this neither conforms with XML Schema, where subtyping is defined by name through restrictions and extensions, nor does it allow a free mix of objects and XML. Further, Xtatic uses pattern matching for XML projections, which fits well with the chosen type system but lacks first-class queries.

In contrast to XJ and Xtatic, Cω does not treat XML as a distinct and separate class. Its ingenuity lies in the uniform integration of the new stream, choice and struct types into the existing types and the generalization of member access— ‘‘the power is in the dot”. In fact, generalized member access in Cω achieves many of the benefits that other type systems try to solve. For example, a long standing problem is how to write a query over data that comes from two sources that are similar, modulo some distribution rules, but not the same [8]. The type algebra of regular expression types often allows a factorization which makes this scenario possible. Generalized member access, on the other hand, handles this problem itself, without the need for distribution rules at the type level.

Another popular approach to deal with XML in an object-oriented language is by using so called data-bindings. A data-binding generates some strongly typed object representation from a given XML schema (XSD). JAXB for Java and xsd.exe in the .NET framework generate classes from a given XSD. However, it is often impossible to generate reasonable bindings, since the rich type system of XSDs cannot adequately be mapped onto classes and interfaces. As a consequence the resulting mappings are often weakly typed.

Cω takes a different but simpler view: XML is considered to be a serialization syntax for the rich type system of Cω. We are not tied to a particular XML data model. While Cω by design doesn’t support the entirety of the full XML stack, in our experience Cω’s type system and language extensions are rich enough to support realistic scenarios. We have written a large number of applications, including the complete set of XQuery Use Cases, several XSL stylesheets, and a substantial application (50KLOC) to manage TV listings.

5 Conclusions and Future Work

In this paper we have considered the problem of manipulating relational and semi-structured data within common object-oriented languages. We observed that existing methods using APIs provide poor support for these common application scenarios.

Therefore, we have proposed a series of elegant extensions to C$^\sharp$ that provides type-safe, first-class access to, and querying of, these forms of data. We also have built a full compiler that implements our design. In this paper we have studied these extensions formally.

This work represents an industrial application of formal methods; on the whole, we found the process of formalizing our intuitions extremely useful, and indeed we managed to trap a number of subtle design flaws in the process. (In addition we had to formalize a fragment of C$^\sharp$, which was a little subtle in places. For example, we believe that this paper gives the first formal operational semantics for iterators.) That said, we also found it useful to be simultaneously developing a compiler. On a small number of occasions we found that our formalization was too high-level, in that it failed to capture some lower-level issues. Also whilst FCω is small enough to prove theorems about by hand, we should have liked to formalized a larger fragment of the language. At the moment, this seems unrealistic without more highly developed machine assistance.

One aspect of this project that we should like to consider further is the compilation. The Common Type System (CTS) for the Common Language Runtime (CLR) whilst general, lacks support for structural types. As our current compiler targets .NET 1.1, this means that the choice and anonymous structs types have to be ‘‘simulated”. In future work, we plan to study extending the CLR with structural types. This would also enable more effective compilation of other languages that offer structural types, such as functional languages. It would also be interesting to study whether the lightweight covariance of Cω could be added to the CTS and other languages.

Implementation Status. A prototype Cω compiler is freely available. It covers the entire safe fragment of C$^\sharp$ and includes all the data access features described in this paper (and more) and also the ‘‘polyphonic” concurrency primitives [3]. (Available from `http://research.microsoft.com/comega`.)

References

1. A. Albano, G. Ghelli, and R. Orsini. Types for databases: the Galileo experience. In *Proceedings of DBPL*, 1989.
2. A. Albano, G. Ghelli, and R. Orsini. Fibonacci: A programming language for object databases. *Journal of Very Large Data Bases*, 4(3):403–444, 1995.
3. N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C$^\sharp$. *TOPLAS*, 26(5):769–804, 2004.
4. V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *Proceedings of ICFP*, 2003.
5. G.M. Bierman, M.J. Parkinson, and A.M. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge, 2003.
6. G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to Java. In *Proceedings of OOPSLA*, 1998.
7. V. Breazu-Tannen, T. Coquand, C.A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and computation*, 93(1):172–221, 1991.
8. P. Buneman and B.C. Pierce. Union types for semistructured data. In *Proceedings of IDPL*, 1998.
9. D. Chamberlin et al. XQuery use cases. `http://www.w3.org/TR/xquery-use-cases/`.

10. S. Boag et al. XQuery. `http://www.w3.org/TR/xquery`.
11. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of POPL*, 1998.
12. C. Fournét and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL*, 1996.
13. V. Gapeyev and B.C. Pierce. Regular object types. In *Proceedings of ECOOP*, 2003.
14. M. Harren, M. Raghavachari, O. Shmueli, M. Burke, V. Sarkar, and R. Bordawekar. XJ: Integration of XML processing into Java. Technical report, IBM Research, 2003.
15. H. Hosoya and B.C. Pierce. XDuce: A typed XML processing language. In *Proceedings of WebDB*, 2000.
16. H. Hosoya, J. Vouillon, and B.C. Pierce. Regular expression types for XML. In *Proceedings of ICFP*, 2000.
17. M. Howard and D. LeBlanc. *Writing Secure Code*. Microsoft Press, 2003.
18. A. Igarashi, B.C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
19. D. Leijen and E. Meijer. Domain Specific Embedded Compilers. In *Proceedings of Conference on Domain-Specific Languages*, 1999.
20. F. Matthes, S. Müßig, and J.W Schmidt. Persistent polymorphic programming in Tycoon: An introduction. Technical report, University of Glasgow, 1994.
21. E. Meijer, W. Schulte, and G.M. Bierman. Programming with circles, triangles and rectangles. In *Proceedings of XML*, 2003.
22. E. Meijer, W. Schulte, and G.M. Bierman. Unifying tables, objects and documents. In *Proceedings of DP-COOL,*, 2003.
23. B.C. Pierce. *Types and programming languages*. MIT Press, 2002.
24. J. Price. *Java programming with Oracle SQLJ*. O'Reilly, 2001.
25. D. Yu, A. Kennedy, and D. Syme. Formalization of generics for the .NET common language runtime. In *Proceedings of POPL*, 2004.

Prototypes with Multiple Dispatch: An Expressive and Dynamic Object Model

Lee Salzman and Jonathan Aldrich

Carnegie Mellon University, Pittsburgh, PA 15217, USA
lsalzman@alumni.cmu.edu
jonathan.aldrich@cs.cmu.edu

Abstract. Two object-oriented programming language paradigms—dynamic, prototype-based languages and multi-method languages—provide orthogonal benefits to software engineers. These two paradigms appear to be in conflict, however, preventing engineers from realizing the benefits of both technologies in one system. This paper introduces a novel object model, prototypes with multiple dispatch (PMD), which seamlessly unifies these two approaches. We give formal semantics for PMD, and discuss implementation and experience with PMD in the dynamically typed programming language Slate.

1 Overview

We begin the paper by describing a motivating example that shows the limitations of current, popular object-oriented languages for capturing how method behavior depends on the interaction between objects and their state. The example shows that multi-methods can cleanly capture how behavior depends on the interaction between objects, while dynamic, prototype-based languages can cleanly capture how behavior depends on object state. Unfortunately, unifying highly dynamic, prototype-based languages with multi-methods is hard, because traditional multi-methods assume a static class hierarchy that is not present in dynamic prototype-based languages.

In section 3 we describe Prototypes with Multiple Dispatch (PMD), an object model that combines the benefits of dynamic, prototype-based languages with multi-methods. PMD supports both paradigms by introducing a role concept that links a slot within an object to a dispatch position on a method, and defining a dynamic multi-method dispatch mechanism that traverses the graph of objects, methods, and roles to find the most specific method implementation for a given set of receiver objects.

Section 4 defines the PMD model more precisely using operational semantics. Section 5 demonstrates the expressiveness of PMD through the standard library of Slate, a dynamically-typed language that implements the PMD object model. Section 6 describes an efficient algorithm for implementing dispatch in Slate. Section 7 describes related work, and section 8 concludes.

2 Motivating Example

In this section, we use a simple running example to examine the benefits and limitations of two current trends in object-oriented programming: prototype-based languages and multi-method languages. Objects were originally invented for modeling and simulation purposes, and our example follows this tradition by modeling a simple ocean ecosystem.

Figure 1 presents the running example in a conventional class-based language like Java or C#. The inheritance hierarchy is made up of an abstract `Animal` superclass and two concrete subclasses, `Fish` and `Shark`. An animal's behavior is defined by the `encounter` method. Fish swim away from healthy sharks, but ignore other animals. If a shark is healthy, it will eat any fish it encounters and fight other sharks; if the shark is not healthy it will swim away from other animals. When a shark fights, it becomes unhealthy.

This example illustrates behavior that depends on both an object's class and its state, echoing many important real-world programming situations. For example, a fish's behavior depends on the type of animal that it encounters. A shark's behavior depends both on the type of animal it encounters and its current health.

```
class Animal {
   abstract method encounter (other : Animal);
   method swimAway () { ... }
}

class Fish inheriting Animal {
   method encounter (other : Animal) {
      if (other.isShark())
         if (other.isHealthy())
            swimAway();
   }
}

class Shark inheriting Animal {
   variable healthy : boolean;
   method isHealthy() {
      return healthy;
   }
   method swallow (other : Animal) { ... }
   method encounter (other : Animal) {
      if (isHealthy())
         if (other.isFish())
            swallow (other);
         else if (other.isShark())
            fight (other);
      else
         swimAway();
   }
   method fight (other : Shark) {
      healthy := False;
   }
}
```

Fig. 1. A simple inheritance hierarchy modeling an ocean ecosystem. The encounter method illustrates behavior that depends both on an object's class (Shark or Fish) and its state (healthy or not). In conventional class-based languages, the behavior specification is complex, imperative, and hard to extend with additional classes

In this example, object-oriented programming is beneficial in that it allows us to encapsulate a shark's behavior within the shark code and a fish's behavior within the fish's code. However, it also shows problems with current object-oriented languages. The specification of behavior is somewhat complex and hard to understand–even for this simple example–because the control structure within the `encounter` methods branches on many conditions. The program is also relatively hard to extend with new kinds of animals, because in addition to defining a new subclass of `Animal`, the programmer must add appropriate cases to the `encounter` methods in `Fish` and `Shark` to show how these animals behave when they encounter the new type of animal.

2.1 Multiple Dispatch

A language with multi-methods dispatches on the classes of all the argument objects to a method, rather than on just the class of the receiver. Multiple dispatch is useful for modeling functionality that depends on the type of multiple interacting objects.

Figure 2 shows the ocean ecosystem modeled using multi-methods. Instead of being written as part of each class, multi-methods are declared at the top level

```
class Animal { }
   method swimAway (animal : Animal) { ... }

class Fish inheriting Animal { }
   method encounter (animal : Fish, other : Animal) { }
   method encounter (animal : Fish, other : Shark) {
      if (other.isHealthy())
         swimAway(animal);
   }

class Shark inheriting Animal {
   variable healthy : boolean;
}
   method isHealthy (animal : Shark) {
      return animal.healthy;
   }
   method swallow (animal : Shark, other : Animal) { ... }
   method encounter (animal : Shark, other : Fish) {
      if (animal.isHealthy())
         swallow (animal, other);
      else
         swimAway(animal);
   }
   method encounter (animal : Shark, other : Shark) {
      if (animal.isHealthy())
         fight (animal, other);
      else
         swimAway(animal);
   }
   method fight (animal : Shark, other : Shark) {
      animal.healthy := False;
   }
```

Fig. 2. Modeling the ocean ecosystem using multi-methods. Here, the `encounter` method dispatches on both the first and second arguments, simplifying the control structure within the methods and making the system more declarative and easier to extend

and explicitly include the first (or receiver) argument. Multi-methods dispatch on all argument positions, so that one of four `encounter` methods can be called, depending on whether the two animals are both sharks, both fish, or one of each in either order.

Typically, multiple dispatch is resolved by picking the most specific method that is applicable to all of the arguments, with a subtype relation among classes determining this specificity. For example, if a fish encounters a shark, at least two methods are applicable: the first method defined accepts a fish in the first position and any animal in the second position, but the second is more specific, accepting a fish in the first position but only sharks in the second position. In this case the second method would be invoked because it is more specific.

In cases where two methods are equally specific, languages differ. Languages like Cecil that use symmetric dispatch would signal a *message ambiguous* error [5], while languages like CLOS and Dylan would choose a method by giving the leftmost arguments greater priority whenever the specificities of two methods are compared [2, 9].

The example shows that multiple dispatch has a number of advantages over single dispatch. It is more declarative, concise, and easy to understand, because the control-flow branches within the `encounter` method have been replaced with declarative object-oriented dispatch. It is more extensible, because the system can be extended with new objects and new methods without changing existing objects and methods. These advantages are similar to the advantages that object-oriented programming brings relative to procedural programming.

However, there remain problems with the example, as expressed. It is still awkward to express stateful behavior; this is still represented by the control flow branches inside `encounter` methods. Furthermore, the code describing that unhealthy sharks swim away from all other animals is duplicated in two different `encounter` methods. This redundancy makes the program harder to understand, and creates the possibility that errors may be introduced if the duplicated code is evolved in inconsistent ways.

2.2 Prototype-Based Languages

Prototype-based languages, pioneered by the language Self [17], simplify the programming model of object-oriented languages by replacing classes with prototype objects. Instead of creating a class to represent a concept, the programmer creates an object that represents that concept. Whenever the program needs an instance of that concept, the prototype object is cloned to form a new object that is identical in every way except its identity. Subsequent modifications to the clone diverge from the original and vice versa.

Prototype-based languages also emphasize the step-wise construction of objects over a static and complete description. Methods may be added as new "slots" of an object at any time, and in languages like Self, inheritance relationships may also be changed at any time. This emphasis on incremental construction occurs because objects are now self-sufficient entities that contain behavior

```
object Animal;
object Fish;
object Shark;
object HealthyShark
object DyingShark

addDelegation (Fish, Animal);
addDelegation (Shark, Animal);
addDelegation (Shark, HealthyShark);

method Animal.swimAway () { ... }

method Fish.encounter(other) {
   if (other.isHealthyShark())
      swimAway();
}

method HealthyShark.swallow (other : Fish) { ... }
method HealthyShark.fight (other : Shark) {
   removeDelegation(HealthyShark);
   addDelegation(DyingShark);
}

method HealthyShark.encounter (other) {
   if (other.isFish())
      swallow (other);
   else if (other.isShark())
      fight (other);
}
method DyingShark.encounter (other) {
   swimAway();
}
```

Fig. 3. Modeling the ocean ecosystem using a prototype-based language. Here, the health of a shark is modeled by delegation to either the HealthyShark or the DyingShark. These abstractions represent behavior more cleanly and declaratively compared to the solutions described above

as a genuine component of their state, rather than being instances of a class which merely describes their behavior for them.

Figure 3 shows how the ocean ecosystem can be expressed in a prototype-based language. The programmer first creates a prototype Animal object, then creates prototype Shark and Fish objects that delegate to the Animal.

The health of a Shark is represented by delegation to either a HealthyShark object or a DyingShark object. These objects encapsulate the behavior of the shark when it is healthy or dying, respectively. Sharks begin in the healthy state, delegating to the HealthyShark object and thus inheriting its `encounter` method. When a HealthyShark fights, the current object's delegation is changed from HealthyShark to DyingShark, and from that point on the shark inherits the `encounter` method from DyingShark.

This example shows a strength of prototype-based languages: delegation can easily be used to represent the dynamic behavior of an object. The behavior can be changed dynamically when some event occurs simply by changing the object's delegation. Although we use dynamic inheritance as an illustration of the malleability provided by prototype-based languages, other features of these languages provide expressiveness benefits as well. For example, we could just as easily have redefined Shark's `encounter` method in the `fight` method to model changes in health.

Despite the advantages that prototypes bring, some problems remain. Like the original class-based code, the prototype-based implementation of the `encounter` methods branch explicitly on the type of the object being encountered. As discussed earlier, this makes the code more difficult to understand and harder to extend with new kinds of animals.

2.3 Discussion

The advantages of multiple dispatch and prototypes are clearly complementary. Multiple dispatch allows programmers to more declaratively describe behavior that depends on multiple interacting objects. Prototypes allow programmers to more cleanly describe stateful behavior, in addition to other benefits accrued by more malleable objects such as more accessible object representation, finer-grained method definition, and arbitrary object extension.

Because of these complementary advantages, it is natural to suggest combining the two models. Such a combination is difficult, however, because multiple dispatch depends on a predetermined hierarchy of classes, while prototypes generally allow a delegation hierarchy to change arbitrarily at any time.

Thus previous languages such as Cecil that combine these two models restrict delegation and method definition to be relatively fixed at a global scope that may be easily analyzed [5]. Unfortunately, restricting the manipulation of objects and methods, without compensating with additional mechanisms, also eliminates a key advantage of prototypes: the elevation of behavior to state. This fixed delegation hierarchy and method definition becomes reminiscent of classes which also, in general, emphasize this fixed inheritance and construction.

While other techniques for declaratively specifying the dependence of object behavior on state do exist, [6, 8, 13], they are more complex and restricted than dynamic inheritance and method update mechanisms in Self.

3 Prototypes with Multiple Dispatch

The contribution of this paper is describing how a dynamic, prototype-based object model in the style of Self can be reconciled with multiple dispatch. Our object model, Prototypes with Multiple Dispatch (PMD), combines the benefits of these two previous object models.

Figure 4 shows the programmer's view of PMD. The programmer creates an object structure that mirrors the prototype code given earlier. When defining methods, however, the programmer declares all arguments (including the receiver) explicitly, as in the multi-method code given earlier. Instead of giving the class that each argument dispatches on, a prototype object is given.

The code in Figure 4 combines the best of both prototypes and multiple dispatch. As in the prototype case, the behavioral dependence on the health of the shark is modeled as delegation to a HealthyShark or a DyingShark object. This delegation can be changed, for example, if the shark is injured in a fight. At the

```
object Animal;
object Fish;
object Shark;
object HealthyShark;
object DyingShark;

addDelegation (Fish, Animal);
addDelegation (Shark, Animal);
addDelegation (Shark, HealthyShark);

method swimAway (animal : Animal) { ... }

method encounter(animal : Fish, other : Animal) /* A */ { }
method encounter(animal : Fish, other : HealthyShark) /* B */ {
   swimAway(animal);
}

method swallow (animal : Shark, other : Fish) { ... }
method fight (animal : HealthyShark, other : Shark) {
   removeDelegation(animal, HealthyShark);
   addDelegation(animal, DyingShark);
}

method encounter (animal : HealthyShark, other : Fish) /*C*/ {
   swallow (animal, other);
}
method encounter (animal : HealthyShark, other : Shark) /*D*/ {
   fight (animal, other);
}
method encounter (animal : DyingShark, other : Animal) /*E*/ {
   swimAway(animal);
}
```

Fig. 4. Modeling the ocean ecosystem in Prototypes with Multiple Dispatch (PMD). PMD combines multiple dispatch with a dynamic, prototype-based object model, leading to a declarative treatment of both state and dispatch

same time, behavioral dependence on multiple interacting objects is expressed through multiple method declarations, one for each relevant case. In a sense, the code is as clean and declarative as it could possibly be: no state variables or control-flow branches remain.

3.1 Dispatch Model

The key insight that makes PMD work is that multi-methods must be internalized into objects, rather than treated as external entities that dispatch across a fixed inheritance hierarchy. Retaining a largely extrinsic dispatch process, as in previous multi-method languages, inevitably restricts the capability of developers to manipulate the behavior of an object through dynamic inheritance or method update.

In Self, methods are internalized by storing them in slots of the receiver object. PMD cannot use this strategy, however, because a multi-method must operate on multiple objects; there is no distinguished receiver.

We solve this challenge by introducing the concept of the *role* played by a particular object in an interaction defined by a multi-method. Each multi-method defines a role for each of its argument positions. For example, in the last

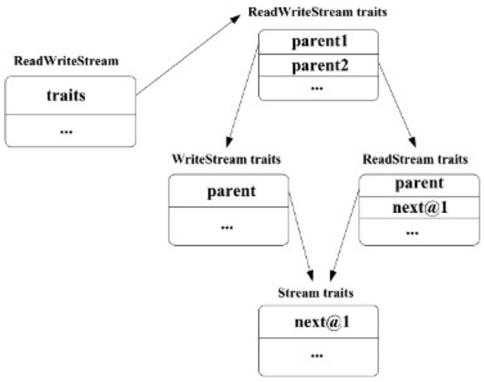


Fig. 5. A conceptual view of prototypes with multiple dispatch

method of Figure 4, the `encounter` method's first role is played by a DyingShark object, while the second role is played by an Animal object.

Each object keeps track of which roles it plays for which multi-methods. Figure 5 shows the roles that different objects play in different `encounter` methods. Animal plays the second role in two different `encounter` method bodies: the ones marked A and E in the code above. Fish plays the first role in the first two methods (since their first parameter dispatches on Fish) and the second role in method C.

Dispatch occurs by searching the delegation hierarchy for inherited methods with the right name and appropriate roles for each of the arguments. For example, consider what happens when a fish encounters a shark that is healthy (i.e., still is delegating to the HealthyShark object). Fish can play the "encounterer" role (role #1) in both methods A and B. The shark can play the "encounteree" role (role #2) in methods A and E, inherited from Animal, method B, inherited from HealthyShark, and method D, defined in the Shark object itself. Only methods A and B will work for both roles. We choose the method to invoke by ordering the delegations for a given object, in case there are multiple such delegations.

A number of semantics are possible for determining the precedence of different applicable methods. The semantics we chose implements a total ordering of methods by considering (first) arguments farther to the left to take precedence over arguments to the right, (second) multiple delegations within a single object to be ordered according to the most recent time of definition, and (third) methods closer in the delegation hierarchy to the supplied method arguments to be more precise.

We chose a total order rather than a partial order, as in Cecil [5], to avoid the possibility of ambiguity in dispatch. A left-to-right ordering is standard, as is the criteria of closeness in the dispatch hierarchy. We chose to prioritize more recent delegations because this gives developers more flexibility to affect the behavior of objects by adding a new delegation. We prioritize first on the argument position, then on the ordering of delegations, then on the distance in the delegation hierarchy, because this gives us a convenient depth-first search algorithm to find the appropriate method (Section 6.1). This algorithm is both more efficient and easier for a programmer to understand than a breadth-first algorithm that would otherwise be required.[1]

4 Formal Model

This section provides a formal model of prototypes with multiple dispatch through a new object calculus. Our calculus borrows ideas from several previous object calculi, but the differences between PMD and previous object models are too great to use a straightforward extension of a previous calculus. For example, only one previous calculus that we know of supports imperative updates of methods [1]. However, this calculus, like most others [3], compiles away delegation by simply copying methods from the delegatee into the delegator. This strategy cannot possibly work in PMD because delegation can change after an object is created. Thus, the representation of objects in the calculus must maintain information about delegation to support this properly.

The most significant difference with all previous calculi is our modeling of multiple dispatch through roles on objects; this makes the common approach of modeling objects as records of methods inappropriate [10, 1, 3]. Although a few object calculi model multi-methods [14, 4], they all model multi-methods as external functions that dispatch over a fixed dispatch hierarchy, while PMD allows the developer to change the methods that are applicable to an object, as well as to modify the inheritance hierarchy at run time.

We instead sketch an untyped, imperative object calculus, PMD (Prototypes with Multiple Dispatch), that precisely describes the semantics of our proposed object model. The main contributions of the model are formalizing multi-method dispatch based on roles, and exposing the choices language designers have for determining dispatching strategies. We hope that the calculus can be extended with a type system, but this remains challenging future work.

4.1 Syntax

Figure 6 explains syntax of PMD. This syntax provides lambda expressions for defining method bodies, object construction through ordered delegation and method definition, and roles that define the various connections between objects. As in Smalltalk [11], method selectors are themselves objects and can be computed.

As PMD is an imperative calculus, the model further assumes a store mapping a store location, used to represent object identity, to an object's representation. The object representation consists first of a sequence of locations denoting the

[1] If depth in the delegation hierarchy were considered first, for example, then simply adding an extra layer of delegation would affect dispatch, which seems extremely counterintuitive.

$l \in locations$	possible object identities in the store
$f ::= \lambda\overline{x}.e$	lambda expressions defining a method body
$e ::= x$	bindings
$\mid l$	locations that the store maps to objects
$\mid e_s(\overline{e})$	invokes method identified by selector e_s upon arguments $\overline{e}$
$\mid e_s(\overline{e}) \leftarrow f$	defining a method at selector e_s with body f, dispatching on $\overline{e}$
$\mid clone(e)$	copies an object
$\mid e \triangleright e_d$	updates e to delegate to e_d
$\mid e \not\triangleright$	removes the last delegation that was added to e
$v ::= l$	reduced values
$d ::= l$	delegations
$r ::= (l, i, f)$	roles contain a method selector, a method parameter index, and a method body
$O ::= (\langle\overline{d}\rangle, \{\overline{r}\})$	objects contain a list of delegations and set of roles
$S ::= l \mapsto O$	store mapping object identity to representation

Fig. 6. The syntax of PMD. The notation $\overline{z}$ denotes a syntactic sequence of z

$$\begin{array}{l}
Animal \overset{def}{=} clone(Root)\\
Fish \overset{def}{=} clone(Root)\\
Shark \overset{def}{=} clone(Root)\\
HealthyShark \overset{def}{=} clone(Root)\\
DyingShark \overset{def}{=} clone(Root)\\
Fish \triangleright Animal\\
Shark \triangleright Animal\\
Shark \triangleright HealthyShark\\
encounter(Fish, HealthyShark) \leftarrow \lambda xy.swimAway(x)\\
encounter(Fish, Animal) \leftarrow \lambda xy.x\\
fight(HealthyShark, Shark) \leftarrow \lambda xy.x \not\triangleright \triangleright DyingShark\\
encounter(HealthyShark, Fish) \leftarrow \lambda xy.swallow(x, y)\\
encounter(HealthyShark, Shark) \leftarrow \lambda xy.fight(x, y)
\end{array}$$

Fig. 7. The example scenario represented in the formal model

objects the particular object delegates to and then a set of roles identifying the methods connected to the particular object.

The notation $S[l]$ will be used to denote object representation corresponding to the location l in the store S, and the notation $S[l \mapsto O]$ will be used to denote the store S adjusted to map the location l to the object representation O.

4.2 Example

Figure 7 presents the running example in the PMD calculus. It still retains all of the conciseness and descriptiveness as the original PMD-inspired example and differs little from it, despite being framed in terms of the lower-level calculus. The

PMD semantics sufficiently captures the mechanisms that lead to the minimal factoring of the running example.

The example assumes the existence of a distinguished empty object Root from which blank objects may be cloned as well as sufficient unique objects defined to cover all method selectors used in the example. Otherwise, the example remains a straight-forward translation of the earlier informal PMD example.

4.3 Dynamic Semantics

Figure 8 presents the core dynamic semantics of PMD. These reduction rules take the form $S \vdash e \hookrightarrow e', S'$, to be read as "with respect to a store S, the expression e reduces in one step to e', yielding a new store S'". The reduction rules define method invocation, method definition, object cloning, and delegation addition and removal. The congruence rules define the order of evaluation in the standard way.

The rule **R-Invoke** looks up the body of the most applicable method, with respect to a method selector and a sequence of method arguments, given by the *lookup* function (defined below in Figure 9). The method arguments are then substituted into the lambda expression/method body which is the result. Substitution occurs as in the lambda calculus.

The rule **R-Method** defines a new method body, to be invoked with a given selector, and dispatching on the given set of objects. A new role is added to each object v_i, stating that the object (or any object that delegates to it) can play the ith role in a dispatch on method selector v_s to method body f. The object representations are updated in the store to reflect this, yielding a new store. The first condition ensures this definition is unique to the particular method selector and arguments; there is no other method body defined upon this exact invocation. The expression reduces to the first argument here only to simplify presentation. We omit a rule for method removal for brevity's sake, which would be a straight-forward inversion of this particular rule.

Note that method definition here affects intrinsic properties of the supplied argument objects' representations, rather than appealing to some extrinsic semantic device. This becomes significant in the rule **R-Clone**, which provides the ubiquitous copying operation found in prototype-based languages. To ensure that the copied object, which bears a new location and representation in the store, responds to all method invocations in similar fashion as the original object, the rule only needs to do duplicate the list of delegations and set of roles. This simple duplication of relevant dispatch information in turn simplifies the implementation of these semantics.

The rules **R-AddDelegation** and **R-RemoveDelegation** together manipulate the ordered list of delegations of an object in stack-like fashion. The rule **R-AddDelegation** adds a target object as a delegation to the top of the ordered list of delegations of the origin object. The rule **R-RemoveDelegation** removes the top of this ordered list and returns the removed delegation target. These two particular rules were merely chosen to simplify presentation, and other alternative rules allowing for arbitrary modification of the list are certainly possible.

Reduction Rules

$$\frac{lookup(S, v_s, \overline{v}) = \lambda\overline{x}.e}{S \vdash v_s(\overline{v}) \hookrightarrow [\overline{v}/\overline{x}]\, e, S'}\ \textbf{R-Invoke}$$

$$\frac{\begin{array}{c}\nexists f'\, (\forall_{0\le i\le n}\, (S_0\,[v_i] = (\langle\cdots\rangle, \{\cdots, (s, i, f')\}))) \\ \forall_{0\le i\le n} \begin{pmatrix} S_i\,[v_i] = (\langle\overline{d}\rangle, \{\overline{r}\}) \\ \bigwedge S_{i+1} = S_i\left[v_i \mapsto (\langle\overline{d}\rangle, \{\overline{r}, (v_s, i, f)\})\right]\end{pmatrix}\end{array}}{S_0 \vdash v_s(v_0 \cdots v_n) \leftarrow f \hookrightarrow v_0, S_{n+1}}\ \textbf{R-Method}$$

$$\frac{S\,[v] = O \qquad l \notin dom(S) \qquad S' = S\,[l \mapsto O]}{S \vdash clone(v) \hookrightarrow l, S'}\ \textbf{R-Clone}$$

$$\frac{S\,[v_o] = (\langle\overline{d}\rangle, \{\overline{r}\}) \qquad S' = S\left[v_o \mapsto (\langle\overline{d}, v_t\rangle, \{\overline{r}\})\right]}{S \vdash v_o \triangleright v_t \hookrightarrow v_o, S'}\ \textbf{R-AddDelegation}$$

$$\frac{\begin{array}{c}S\,[v] = (\langle d_0 \cdots d_n\rangle, \{\overline{r}\}) \qquad n \ge 0 \\ S' = S\,[v \mapsto (\langle d_0 \cdots d_{n-1}\rangle, \{\overline{r}\})]\end{array}}{S \vdash v \not\triangleright \hookrightarrow d_n, S'}\ \textbf{R-RemoveDelegation}$$

Congruence Rules

$$\frac{S \vdash e_s \hookrightarrow e_s', S'}{S \vdash e_s(\overline{e}) \hookrightarrow e_s'(\overline{e}), S'} \qquad \frac{S \vdash e_s \hookrightarrow e_s', S'}{S \vdash e_s(\overline{e}) \leftarrow f \hookrightarrow e_s'(\overline{e}) \leftarrow f, S'}$$

$$\frac{S \vdash e_i \hookrightarrow e_i', S'}{S \vdash v_s(v_0 \cdots v_{i-1}, e_i, e_{i+1} \cdots e_n) \hookrightarrow v_s(v_0 \cdots v_{i-1}, e_i', e_{i+1} \cdots e_n), S'}$$

$$\frac{S \vdash e_i \hookrightarrow e_i', S'}{S \vdash v_s(v_0 \cdots v_{i-1}, e_i, e_{i+1} \cdots e_n) \leftarrow f \hookrightarrow v_s(v_0 \cdots v_{i-1}, e_i', e_{i+1} \cdots e_n) \leftarrow f, S'}$$

$$\frac{S \vdash e_o \hookrightarrow e_o', S'}{S \vdash e_o \triangleright e_t \hookrightarrow e_o' \triangleright e_t, S'}$$

$$\frac{S \vdash e_t \hookrightarrow e_t', S'}{S \vdash v \triangleright e_t \hookrightarrow v \triangleright e_t', S'} \qquad \frac{S \vdash e_o \hookrightarrow e_o', S'}{S \vdash e_o \not\triangleright \hookrightarrow e_o' \not\triangleright, S'}$$

Fig. 8. The dynamic semantics of PMD

4.4 Dispatch Semantics

Figure 9 presents the dispatch semantics provided by the *lookup* function. The rule **R-Lookup** is a straight-forward transcription of the idea of multiple dispatch. It states that a method body should be dispatched if it is applicable - a member of the set of applicable methods - and it is the most specific of all such method bodies, or rather, is the least method body according to an operator that compares the applicable method bodies. The *rank* function and $\prec$ operator together implement this comparison operator.

$$\frac{f \in applic(S, v_s, \overline{v}) \qquad \forall_{f' \in applic(S, v_s, \overline{v})} \left(f = f' \bigvee rank(S, f, v_s, \overline{v}) \prec rank(S, f', v_s, \overline{v})\right)}{lookup(S, v_s, \overline{v}) = f} \textbf{ R-Lookup}$$

$$applic(S, v_s, v_0 \cdots v_n) \overset{def}{=} \left\{ f \middle| \forall_{0 \le i \le n} \begin{pmatrix} delegates(S, v_i) = \langle d_0 \cdots d_m \rangle \bigwedge \\ \exists_{0 \le k \le m} \left(S\left[d_k\right] = \left(\left\langle \overline{d'} \right\rangle, \{\cdots, (v_s, i, f)\}\right)\right) \end{pmatrix} \right\}$$

$$rank(S, f, v_s, v_0 \cdots v_n) \overset{def}{=} \prod_{0 \le i \le n} \min_{0 \le k \le m} \left\{ k \middle| \begin{array}{c} delegates(S, v_i) = \langle d_0 \cdots d_m \rangle \\ \bigwedge S\left[d_k\right] = \left(\left\langle \overline{d'} \right\rangle, \{\cdots, (v_s, i, f)\}\right) \end{array} \right\}$$

Fig. 9. The dispatch semantics of PMD

We then define the *applic* set of methods as those methods for which every argument either contains a satisfactory role for the method, or delegates to an object with such a role. A role here is satisfactory if index of the method argument on which it is found matches that in the role, and the method selector matches that in the role as well. This definition relies on the *delegates* function, which returns an ordered list of all delegated-to objects transitively reachable by the delegation lists in objects, and including the original method argument itself. In the case of a statically-fixed delegation hierarchy, this rule exactly mirrors the applicability criteria in previous multi-method languages such as Cecil, Dylan and CLOS.

Note that because of the first condition of **R-Method**, only one method body can be directly defined on a tuple of objects at a particular selector. Thus, in the absence of delegation, dispatch is trivial since the applicable set of methods contains at most a single method body. Ranking the applicable methods is thus a necessary consequence of delegation.

Finally, the *rank* function, conceptually, finds, among those methods in the *applic* set, the distance at which the roles corresponding to some method appeared. Given the ordered list of *delegates* for an argument described above, it determines the minimal position at which a delegated-to object contains a satisfactory role corresponding to the method. The $\prod$ operator, which also parameterizes the *rank* function, combines these minimal positions for each argument and produces a single rank value - conceptually, a n-dimensional coordinate in the ranking. The total ordering given by the *delegates* function facilitates the ordering that *rank* provides, without which these semantics would be trickier to define.

We assume here that a particular method body is unique to a single method definition. So, in the absence of the rule **R-Clone**, for a specific method selector and parameter index, there can only exist a single satisfactory role corresponding to that particular method body. However, since we do include **R-Clone**, and a role may thus be copied to another object, multiple satisfactory roles corresponding to the method body exist and the closest role among them in the *delegates* ordering is chosen.

We leave the *delegates* function, the $\prec$ operator, and the $\prod$ operator undefined. The reader may define these arbitrarily to suit their intended dispatch semantics. Slate's semantics for these operators are defined along with Slate's dispatch algorithm in Section 6.1.

5 Slate

Prototypes with Multiple Dispatch has been implemented in Slate [16], a dynamically typed programming language. Self [17], Cecil [5], and CLOS [2] directly inspired the design of Slate and the PMD model on which it is based. However, due to the retained flexibility of prototypes in PMD, Slate most strongly resembles Self and retains much of its language organization without greatly compromising its simple object model.

The following section provides a taste of the Slate language through examples from the Slate standard library illustrating the benefits of the PMD model.

5.1 Brief Overview

The syntax and system organization of Slate strongly resembles that of Self and Smalltalk [11]. Due to space limitations, we omit a detailed discussion of Slate's syntax, which should be understandable to the reader familiar with these languages; the syntax is also documented in detail elsewhere [16].

We briefly describe Slate's method definition syntax, which is the primary syntactic difference between Slate and Self or Smalltalk. A method definition looks like a method send in Self or Smalltalk, except that one or more of the arguments[2] is qualified by the object on which the method dispatches. The qualified argument syntax is of the form "`parameterName @ roleArgument`", with "`roleArgument`" identifying the object associated with this role of this method body, and "`parameterName`" being a variable bound in the body of the method. The "`roleArgument`" can be omitted, in which case the method dispatches on the distinguished object "`Root`" to which most other objects in Slate delegate. The presence of at least one parameter with a role argument qualifier is what signals a method definition in the grammar (as opposed to a method invocation).

Some important messages to be used in the subsequent examples include:

resend Resends the message that invoked the current method while ignoring any methods of greater or equal precedence in the dispatch order than the current method during dispatch.

prototype clone Returns a new copy of "`prototype`" that contains all the same slots, delegation slots, and roles.

object addSlot: name valued: initialValue Adds a new slot to "`object`" and defines the method "`name`" with which to access its value and the method "`name:`" with which to set it. "`name`" must evaluate to a symbol. The slot's value is initially "`initialValue`".

[2] Note that in a language with multiple dispatch, the "arguments" to a method include the receiver.

object addDelegate: name valued: initialValue This method behaves exactly like "`addSlot:valued:`", except only that the slot is treated as a delegation slot. The value of the delegation slot is treated as an object that "`object`" delegates to.

object traits Accessor message for the "`traits`" delegation slot, which holds a class-like object sharing method roles for a whole family of "clone"-d objects.

block do Evaluates "`block`".

collection do: block Evaluates "`block`" with each element of collection "`collection`" supplied in turn as an argument.

cond ifTrue: trueBlock ifFalse: falseBlock Evaluates "`trueBlock`" if "`cond`" evaluates to "`True`", or instead "`falseBlock`" if it evaluates to "`False`".

5.2 Example: Instance-Specific Dispatch

Instance-specific dispatch is an extensively used idiom in Slate, benefiting from its prototype-based model. When combined with multiple dispatch, it begins to strongly resemble pattern-matching [15] while still within an object-oriented framework. For example, much of the boolean logic code in Slate is written in a strikingly declarative form using instance-specific dispatch:

```
_@True and: _@True [True].
_@(Boolean traits) and: _@(Boolean traits) [False].
_@False or: _@False [False].
_@(Boolean traits) or: _@(Boolean traits) [True].
```

The code dispatches directly on "`True`" and "`False`" to handle specific cases. It then defines methods on "`Boolean traits`" to handle the remaining default cases.

5.3 Example: Eliminating Double Dispatch

Smalltalk [11] and similar languages based on single dispatch typically rely on an idiom called "double dispatch" to work around the limitations this model imposes. In this idiom, a method dispatches on the receiver first, then invokes a helper method (whose name encodes the receiver's class) on the argument which provides a second dispatch.

Double dispatch frequently surfaces in such places as Smalltalk numerics system, making the code more inefficient and harder to understand and extend compared to optimized multi-method dispatch. For example, when a new kind of number is added to the system, all the double dispatch code for arithmetic, distributed among many diverse classes, must be updated to take the new type of number into account.

Slate's native support for multiple dispatch avoids these problems. It is relatively simple to extend Slate's numerics system while keeping these extensions well-encapsulated and without needing global changes to other objects. For example, the following code illustrates how an epsilon object, a negligibly small yet non-zero value, may be integrated into Slate's library in a straightforward and modular way:

```
numerics addSlot: #PositiveEpsilon valued: Magnitude clone.

_@PositiveEpsilon isZero
[False].
_@PositiveEpsilon isPositive
[True].
x@(Magnitude traits) + _@PositiveEpsilon
[x].
```

It is also common in Smalltalk to find many methods such as "`asArray`" or "`asDictionary`" for converting a certain object to the type indicated by the message name. This results in a unnecessary proliferation of related messages and is effectively a manual encoding of the double dispatch idiom.

With the aid of PMD, Slate can support a more expressive and uniform protocol for coercing objects of one type to another via the message "`as:`". The object to convert is supplied along with an instance (as opposed to a class) of some object type the programmer would like the original to coerce to. To define coercions, the programmer need only define a particular method for her new objects as in the following code:

```
x@(Root traits) as: y@(Root traits)
[(x isSameAs: y)
  ifTrue: [x]
  ifFalse: [x conversionNotFoundTo: y]
].
c@(Collection traits) as: d@(Collection traits)
[d newWithAll: c].
s@(Sequence traits) as: ec@(ExtensibleCollection traits)
[| newEC |
  newEC: (ec newSizeOf: s).
  newEC addAll: s.
  newEC
].
s@(Symbol traits) as: _@(String traits)
[s name].
```

5.4 Example: Supporting System Reorganization

Another benefit of using a prototype object system as the language core is that it easily supports reorganizing the language to support new features or remodel old ones.

For instance, Slate uses a depth-first search strategy for finding roles on delegated-to objects. Whichever roles are found first according to this order take precedence over ones found later. However, this simplistic scheme, while allowing an efficient dispatch algorithm and providing the illusion of single inheritance, easily becomes inappropriate in the presence of multiple inheritance.

Figure 10 illustrates the problem. A `ReadWriteStream` is derived from both a `WriteStream` and a `ReadStream`, and so its traits object delegates to both of their traits objects as well. `ReadStream`, in particular, might override a method "`next`" on the basic `Stream` prototype. However, should the dispatch algorithm visit `ReadWriteStream`'s traits, `WriteStream`'s traits, `ReadStream`'s traits, and `Stream traits` in that order, the "`next`" method on `Stream` will resurface and take precedence over the version on `ReadStream`. Ideally, the search should

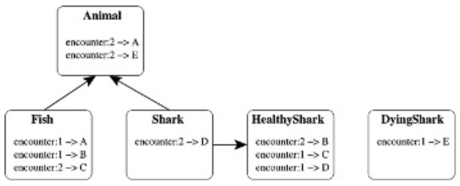


Fig. 10. Slate's original traits inheritance model. Multiple inheritance occasionally results in problems with sequencing methods

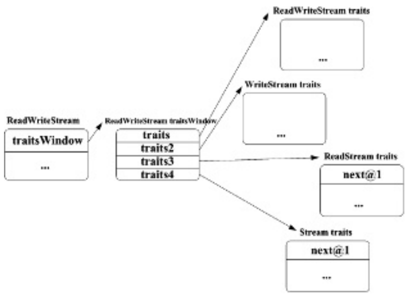


Fig. 11. Slate's new traits inheritance model allows the more desirable breadth-first sequencing of methods

proceed breadth-first, so that `Stream`'s traits object is visited only after both `WriteStream`'s and `ReadStream`'s traits objects have been visited.

This behavior became severely confusing at times, yet merely throwing an error in this case forces the programmer to manually disambiguate it by defining a new method. Instead of adding a barrier to object reuse, a simple reorganization of the traits object system, still only replying upon native delegation and dispatch of PMD, allowed a much more satisfactory behavior.

Instead of having traits directly delegate to their parent traits, an extra layer of indirection was added in the form of a traits window. Objects now delegate to this window instead of directly to traits, and this window merely keeps a list of traits in the exact order they should be visited by the dispatch algorithm. This has the added benefit that even orderings that are expensive to compute and might otherwise defeat various optimizations Slate uses to enhance the performance of dispatch may be freely used and customized at a whim without any negative impacts. Figure 11 illustrates this new organization.

Yet, because Slate is based upon a prototype object system, this did not require any profound changes to the language's implementation to effect this new organization. Most of the changes were localized within the standard library itself, and mostly to utility methods used to construct new prototypes. Only a few lines of code in the interpreter itself that depended on the structure of certain primitively provided objects needed revision.

5.5 Example: Subjective Dispatch

In earlier work on the Self extension Us [18], it was noted that any language with multiple dispatch can easily implement subjective dispatch similar to that provided by Us. In this view of subjective dispatch, a subject is merely an extra implicit participant in the dispatch process, supplied in ways other than directly via a message invocation.

As PMD provides multiple dispatch, Slate supports subjective dispatch of this sort with only slight changes to its semantics. It maintains a distinguished subject in its interpreter state, which is implicitly appended to the argument lists of message invocations and method definitions whenever either is evaluated within this subject. The interpreter also provides a primitive message "`changeSubject:`" to modify the current subject. The semantics of Slate otherwise remain unchanged; programs which do not use subjects are not affected, as all methods dispatch on the default subject.

Further, prototypes naturally support composition of subjects by delegation, allowing for a sort of dynamic scoping of methods by merely linking subjects together with dynamic extent. The message "`seenFrom:`" is easily implemented to this effect:

```
addSlot: #Subject valued: Cloneable clone.
Subject addDelegate: #label.
Subject addDelegate: #previousSubject.

m@(Method traits) seenFrom: label
[| newSubject |
  newSubject: Subject clone.
  newSubject label: label.
  newSubject previousSubject: (changeSubject: newSubject).
  m do.
  changeSubject: newSubject previousSubject
].
```

This subjective behavior can easily allow for the implementation of cross-cutting aspects of a program, implementing behavior similar to `cflow` in AspectJ [12]. The following code illustrates this through the implementation of an undoable transaction, which works by intercepting any modifications to objects via the message "`atSlotNamed:put`", logging the original value of the slot, then allowing the modification to proceed:

```
addSlot: #Transaction valued: Cloneable clone.
Transaction addSlot: #undo valued: ExtensibleArray newEmpty.
Transaction addSlot: #replay valued: ExtensibleArray newEmpty.

t@Transaction log: object setting: slot to: newValue
[| oldValue |
  oldValue: (object atSlotNamed: slot).
  t undo addLast: [object atSlotNamed: slot put: oldValue].
  t replay addLast: [object atSlotNamed: slot put: newValue].
].
t@Transaction undo
[t undo reverseDo: [| :action | action do]].
t@Transaction replay
[t replay do: [| :action | action do]].
```

```
[object atSlotNamed: slot@(Symbol traits) put: value
  [Transaction log: object setting: slot to: value.
    resend
  ].
] seenFrom: Transaction.
```

6 Dispatch in Slate

Many optimizations have been explored to enhance the performance of programs in Slate. This section details implementation strategies used in Slate and that may be applied to implementations of PMD for other languages.

6.1 Dispatch Algorithm

The formalization presented in section 4 leaves open a number of practical considerations about how to implement the core dispatch algorithm of PMD. These issues include determining the proper order of delegations, the candidate set of methods that may be applicable, and finally, the ranks of these methods and how to represent them. Various optimizations also expediently reduce the memory and processing requirements of the algorithm.

The programming language Slate serves as a canonical implementation of PMD and utilizes a dispatch algorithm geared toward a lexicographic ordering

```
dispatch(selector, args, n) {
  for each index below n {
    position := 0
    push args[index] on ordering stack
    while ordering stack is not empty {
      arg := pop ordering stack
      for each role on arg with selector and index {
        rank[role's method][index] := position
        if rank[role's method] is fully specified {
          if no most specific method
             or rank[role's method]  ≺  rank[most specific method] {
            most specific method := role's method
          }
        }
      }
      for each delegation on arg {
        push delegation on ordering stack if not yet visited
      }
      position := position + 1
    }
  }
  return most specific method
}
```

Fig. 12. Pseudo-code for the basic dispatch algorithm used in Slate

of methods and a number of optimizations, including efficient encoding of rank vectors, sparse representation of roles, partial dispatch, and method caching. Slate's dispatch algorithm guides and motivates the subsequent implementation discussion.

Figure 12 outlines in pseudo-code a basic version of the dispatch algorithm. The comparison operator $\prec$ is as in the formalism and may be chosen to implement either a partial or lexicographic ordering as desired, the latter of which is used in Slate. The order in which delegations from a given object are pushed onto and popped from the ordering stack, analogous to the *delegates* function in the formalism, determines the ordering under multiple delegation and should be chosen as is applicable to the implementation. A simple vector of positions in a rank here provides the $\prod$ operator of the formalism. If one overlooks the necessary bookkeeping for rank vectors, this algorithm strikingly resembles the message lookup algorithm utilized by Self.

The process for constructing a depth-first ordering of delegations is straightforward. One maintains a stack of visited but not yet ordered objects from which elements of the ordering are drawn. If the host language allows cyclic delegation links, one also need maintain a set of objects already visited, easily represented by marking the objects directly, to avoid traversing the same delegation twice. If one further assumes object structure is represented by maps, as in Self, or classes, this visited set may be stored on a per-map or per-class basis without loss. The stack is then processed by popping objects off the top, assigning them the next position in the ordering, and then pushing all their delegations onto the stack unless they were already visited.

Role information is stored directly on the objects themselves (or their map or class) and each role identifies a potentially applicable method, or rather, a method that is supported by at least one of the arguments to the method invocation. One may conveniently collect all the candidate methods and their ranks while determining the delegation ordering, merely traversing an object's roles, for the given argument position and method selector, as it is popped off the ordering stack. An auxiliary table, which may be cheaply distributed among the methods themselves, stores the currently determined rank vector of the method, augmenting the method invocation argument's respective component of the rank vector with the current position in the delegation ordering. When a method's rank becomes fully determined, the method is noted as the most specific method (found so far) if its rank is less than the previously found most specific method, or if it is the first such method found. Once the delegation stack has been fully processed for each method invocation argument, the resulting most specific method, if one exists, is a method whose rank is both minimal and fully specified at all argument positions.

6.2 Rank Vectors

One may represent rank vectors themselves efficiently as machine words, with a fixed number of bits assigned to each component up to some fixed number of components. If one assumes method arguments have lexicographical ordering, then simple integer comparisons suffice to compare ranks, where more significant components are placed in more significant bits of the integer represented in the machine word. However, if one assigns each component of the rank number a fixed number of representation bits and if the rank vectors themselves are fixed size, the maximum length of a delegation ordering that may be reflected in each component is also effectively fixed as well as the maximum number of method parameters. One need only provide a fall-back algorithm using arbitrary precision rank vectors in case the ordering stack overflows or if an excessive number of arguments are present at a method invocation. Anecdotally, the majority of methods contain small numbers of parameters and inheritance hierarchies (and similarly delegation hierarchies) are small, so this fall-back algorithm is rarely necessary, if ever.

6.3 Sparse Representation of Roles

In Slate, the delegation hierarchy is rooted at one specific object so that certain methods may be defined upon all objects. However, since this object always assumes the bottom position in the delegation ordering, any roles defined upon it will always be found and always be the least specific such roles with respect to other roles with the same method selector and argument position. These roles do not aid in disambiguating the specificity of a given method since they occupy the bottom of the ordering and, in effect, contribute no value to the rank vector.

The majority of methods in the Slate standard library dispatch on the root object in most arguments positions, so representing these roles needlessly uses memory and adds traversal overhead to the dispatch algorithm. In the interests of reducing the amount of role information stored, one need not represent these roles if one identifies, for each method, the minimum set of roles that need be found for a rank vector to be fully specified and so allows the size of this set of roles to be less than the number of actual method parameters. This set of roles does not contain any roles specified on the root object. A method is now applicable when this minimum set of roles is found during dispatch, rather than a set of roles corresponding to all method parameters. In the interests of reducing duplication of information, Slate stores information about the size of this minimum set of roles on the method object linked by these roles.

6.4 Partial Dispatch

Because of Slate's sparse representation of roles, the dispatch algorithm may determine a method to be applicable, or rather, its minimal set of roles may be found, before it has finished traversing the delegation orderings of all argument positions. The basic algorithm, however, requires that the entire delegation ordering of all arguments be scanned to fully disambiguate a method's specificity and ensure it is the most specific. The majority of methods in the Slate standard library not only dispatch on fewer non-root objects than the number of method parameters, but only dispatch on a single non-root object, and are, in effect, only singly polymorphic. Scanning the entire delegation orderings for all objects under such conditions is wasteful and needless if an applicable method is unambiguously known to be the most-specific method and yet dispatch still continues.

The key to remedying this situation is to take advantage of Slate's lexicographic ordering of method arguments and also note that a role not only helps identify an applicable method, but a role also indicates that some method is possibly applicable in the absence of information about which other roles have been found for this method. If no roles corresponding to a method are found, then the method is not applicable. If at least at least one role corresponding to a method is found, then this method may become applicable later in the dispatch and effect the result should its determined rank vector precede the rank vectors of other applicable methods.

Dispatch in Slate traverses method arguments from the lexicographically most significant argument to the least significant argument. So, for any role found, its contribution to the rank vector will necessarily decrease with each successive argument position traversed. If some method is known to be the most specific applicable method found so far, and a role for a contending method is found whose contribution to its respective rank vector would still leave it less specific than the most specific method, then no subsequent roles found for the contending method will change the method result as they contribute lexicographically less significant values. Thus, one only need maintain the partial rank vector, representing the contention for most specific method, corresponding to the lexicographically most significant roles found up to the current point of traversal. If any applicable method's rank vector precedes this partial rank vector, then it is unambiguously the most specific method, since there are no other more specific methods that may later become applicable.

For example, if one method singly dispatches on the Shark prototype, and another similarly named method dispatches on the Animal prototype in a lexicographically less significant or equally significant argument position, then dispatch will determine the Shark prototype's method to be applicable as soon as the Shark prototype is traversed and before traversing the Animal prototype. If no other roles were found at lexicographically more significant positions, or on preceding objects in the delegation ordering for the lexicographically equal argument position, then there is no possible contention for the resulting most specific method, and the Shark prototype's method must be the most specific.

Intriguingly, this optimization reduces the cost of dispatch to the amount of polymorphism represented in the entire set of candidate methods. So, if all methods only dispatch on their first argument, the dispatch algorithm effectively degenerates to a traditional single dispatch algorithm and need never examine more than the first argument or traverse farther down the delegation hierarchy than where the first candidate method is found. The algorithm then only incurs the cost of maintaining the rank information above the cost of single dispatching. Single dispatching becomes a special-case optimization of the PMD dispatch semantics. Further, almost all the dispatches in the Slate standard library were found to terminate early due to this optimization, rather than requiring a full traversal. This number closely corresponds to the fraction of methods dispatching on fewer non-root objects than their number of arguments, which supports this intuition.

6.5 Method Caching

Various global and inline method caching schemes may be extended to fit the dispatching algorithm and provide an essentially constant time fast-path for method invocation under PMD. Given partial dispatching and if for each method selector one identifies the global polymorphism of the set of methods it identifies (the set of argument positions any roles have been specified in), one only need store the significant arguments positions, as given by the global polymorphism, as the keys of the cache entries. However, cache entries must still have a capacity to store up to the maximally allowable amount of polymorphism for caching. In the degenerate case of global polymorphism of only the first argument, this extended caching scheme degenerates to an ordinary single dispatch caching scheme. The method caching optimization assumes that there are no changes to delegation relationships and no method addition or removal; if these changes are made, the caches must be invalidated in the general case.

7 Related Work

Section 4 described related work in the area of formal object models. Three programming languages significantly influenced the development of PMD and the implementation in Slate: Self [17], CLOS [2], and Cecil [5].

Self attempted to provide a Smalltalk [11] better suited for interactive programming and direct manipulation of objects by dispensing with classes and providing self-representing objects. These objects simply contain slots - modifiable, named value-holders - which can serve as ordinary bindings, method definitions, or delegations. Further, objects are used to more flexibly represent traditionally fixed implementation facilities such as namespaces, shared behavior, and user interfaces. Slate, to date, borrows and benefits from much of this system organization while expanding upon the notion of method definition as in PMD.

CLOS [2] is an extension to Common Lisp that provides object-oriented programming through classes and generic functions. Generic functions are functions made up multiple method cases, which a multiple dispatch algorithm chooses among by examining the classes of all the arguments to a method call. A subtyping relation between the classes of parameters and arguments determines the applicable method bodies and their relative specificities. CLOS linearly (totally) orders both the class and method. The class hierarchy is sequenced into a precedence list to disambiguate any branches in the hierarchy as a result of multiple inheritance. Leftmost parameters also take precedence over the rightmost, disambiguating cases where not all the parameter classes of one method are subtypes of the respective parameter classes of another. The formalism of PMD borrows the idea of a total ordering of inheritance and method arguments in its dispatch semantics to avoid appealing to subtyping, but dispenses with classes and the extrinsic notion of generic functions.

Dylan [9] is another dynamically-typed object-oriented language with multi-methods. Like CLOS, it gives precedence to the leftmost parameter of a function during multi-method dispatch.

Cecil [5] is the first language known by the authors to integrate a prototype-inspired object model with multiple dispatch. Cecil dispenses with the slot-based dynamic inheritance of Self, opting instead to fix delegation between objects at the time an object is instantiated. Method definition is similarly limited to a global scope, restricting certain higher-order uses. Cecil provides multiple dispatch by a form of subtyping upon this relatively fixed delegation hierarchy. This multiple dispatch only provides a partial ordering among objects and method arguments. Dispatch ambiguities arise from the use of multiple delegation or incomplete subsumption among the methods according to the subtyping relation. Such ambiguities raise an error when encountered, and recent work has focused on finding these ambiguities statically [14].

Instead of the slot-based dynamic inheritance of Self, however, Cecil provides predicate classes [6] wherein a fixed delegation relationship is established to a predicate class that is qualified by a predicate. When the predicate of a predicate class is satisfied for some object delegating to it, the object delegating to it will inherit its behavior. When this predicate is not satisfied, this behavior will not be inherited. Predicate dispatch is thus ideal for capturing behavior that depends on a formula over the state of the object, while the dynamic delegation mechanism in PMD captures behavior changes due on program events more cleanly. More recently, predicate classes have been generalized to a predicate dispatch [8, 13] mechanism which unifies object-oriented dispatch with pattern-matching in functional programming languages [15].

Another alternative to dynamic inheritance is changing the class of an object dynamically, as in the Fickle system [7]. This solution is somewhat less expressive than dynamic inheritance, but can be statically typechecked.

One other closely related system is Us, an extension of Self to support subject-oriented programming [18]. Subject-oriented programming allows a method to behave differently depending on the current *subject* in scope. Intuitively, subject-oriented programming can be modeled as an additional layer of dispatch, and multiple dispatch is a natural mechanism for implementing this concept, especially when combined with flexible objects with which to dynamically compose subjects, as the authors of Us noted. However, as Us extends a language only providing single-dispatch, the authors of Us instead chose to separate objects into pieces, all addressed by a single identity. Dynamically composable layer objects implicitly select which piece of the object represents it, effectively implementing a specialized form of multiple dispatch only for this extension. Since PMD provides multiple dispatch and dynamic inheritance, it naturally supports subjects with only a bit of syntactic sugar.

8 Conclusion

This paper introduced a new object model, Prototypes with Multiple Dispatch, that cleanly integrates prototype-based programming with multiple dispatch. The PMD model allows software engineers to more cleanly capture the dynamic interactions of multiple, stateful objects.

Acknowledgments

We thank William Cook, Aaron Greenhouse, Jonathan Moody, and the anonymous reviewers for their feedback on earlier drafts of this material. This work was supported by the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298, NSF grant CCR-0204047, and the Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems."

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.
2. D. G. Bobrow, L. G. DiMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System Specification. In *SIGPLAN Notices 23*, September 1988.
3. V. Bono and K. Fisher. An Imperative, First-Order Calculus with Object Extension. In *European Conference on Object-Oriented Programming*, 1998.
4. G. Castagna, G. Ghelli, and G. Longo. A Calculus for Overloaded Functions with Subtyping. In *Lisp and Functional Programming*, 1992.
5. C. Chambers. Object-Oriented Multi-Methods in Cecil. In *European Conference on Object-Oriented Programming*, July 1992.
6. C. Chambers. Predicate Classes. In *European Conference on Object-Oriented Programming*, 1993.
7. S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More Dynamic Object Reclassification: Fickle II. *Transactions on Programming Languages and Systems*, 24(2):153–191, 2002.
8. M. D. Ernst, C. S. Kaplan, and C. Chambers. Predicate Dispatching: A Unified Theory of Dispatch. In *European Conference on Object-Oriented Programming*, 1998.
9. N. Feinberg, S. E. Keene, R. O. Mathews, and P. T. Withington. *Dylan Programming*. Addison-Wesley, Reading, Massachusetts, 1997.
10. K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.
11. A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison-Wesley, Reading, Massachusetts, 1989.
12. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *European Conference on Object-Oriented Programming*, June 2001.
13. T. Millstein. Practical Predicate Dispatch. In *Object-Oriented Programming Systems, Languages, and Applications*, 2004.
14. T. Millstein and C. Chambers. Modular Statically Typed Multimethods. *Information and Computation*, 175(1):76–118, 2002.
15. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997.
16. B. Rice and L. Salzman. The Slate Programmer's Reference Manual. Available at http://slate.tunes.org/progman/, 2004.
17. D. Ungar and R. B. Smith. Self: The Power of Simplicity. In *Object-Oriented Programming Systems, Languages, and Applications*, pages 227–242. ACM Press, 1987.
18. D. Ungar and R. B. Smith. A Simple and Unifying Approach to Subjective Objects. *Theory and Practice of Object Systems*, 2(3):161–178, 1996.

Efficient Multimethods in a Single Dispatch Language

Brian Foote, Ralph E. Johnson, and James Noble

Dept. of Computer Science, University of Illinois at Urbana-Champaign,
201 N. Goodwin, Urbana, IL 61801, USA
foote@cs.uiuc.edu, johnson@cs.uiuc.edu
School of Mathematical and Computing Sciences, Victoria University of Wellington,
P.O. Box 600, Wellington, New Zealand
kjx@mcs.vuw.ac.nz

Abstract. Smalltalk-80 is a pure object-oriented language in which messages are dispatched according to the class of the receiver, or first argument, of a message. Object-oriented languages that support multimethods dispatch messages using all their arguments. While Smalltalk does not support multimethods, Smalltalk's reflective facilities allow programmers to efficiently add them to the language. This paper explores several ways in which this can be done, and the relative efficiency of each. Moreover, this paper can be seen as a lens through which the design issues raised by multimethods, as well as by using metaobjects to build them, can be more closely examined.

1 Introduction

The designers of object-oriented languages usually consider multimethods and single dispatch to be competing alternatives. This paper describes a variety of ways to implement multimethods in single-dispatch languages such as Smalltalk. It is not surprising that multimethods can be implemented in Smalltalk, because it is a reflective language that has been extended in many ways. However, it is surprising how well multimethods can work with single dispatch. This paper develops a simple extended syntax that makes it easy to mix multimethods and normal methods. The semantics of multimethods are simple, they have no syntactic or performance cost if they are not used, they interoperate well with Smalltalk's metaobjects, and they are as efficient to execute as comparable hand-written code.

Our results show that there is no inherent conflict between multi-methods and single dispatch, at least for Smalltalk.

Introducing multimethods into a single-dispatch language like Smalltalk raises a range of issues: incorporating multimethods into Smalltalk syntax and the programming environment; implementing multimethods using the reflective facilities without changing the underlying virtual machine; and ensuring that multimethods provide good performance, without incurring additional overhead if they are not used.

This paper makes the following contributions:

– A core language design for multimethods in Smalltalk, demonstrating that a multimethod facility inspired by the CLOS Metaobject Protocol [Bobrow 1998] can be added to Smalltalk in a seamless, backwards compatible manner within the spirit of the language.

- An extensible implementation of the core language design, written in Smalltalk, that uses only the language's reflective features and requires no changes to the Smalltalk virtual machine
- An analysis of the performance of a range of implementations based on our framework, demonstrating that this approach is practical.

2 Multiple Dispatch

Like most object-oriented languages, Smalltalk provides single dispatch: a method call (in Smalltalk referred to as a *message send*) considers the dynamic type of one argument: the class of the object to which the message is sent. For example, consider the classical example of a graphical display system, where `GraphicScreen` and `GraphicPrinter` classes are subclasses of the abstract `GraphicalDisplay` class. The `GraphicalDisplay` class can define a number of messages such as `drawLine`, `drawRectangle`, `fillRectangle`, `drawArc`, and so on; then each subclass can implement these messages to display on a screen or a printer respectively.

This design has objects for the graphical displays but not for the graphical entities themselves. An obvious refinement of this design is then to introduce a further series of classes to represent the graphical objects: an abstract `GraphicalObject` class with `Line`, `Rectangle`, `FilledRectangle`, and `Arc` subclasses. This should allow programmers to simply their programs: code such as `aScreen draw: aRectangle` or `aPrinter draw: aLine` should allow any kind of graphical display to draw any kind of object. The problem is that this draw method requires *multiple dispatch*— the method body to be invoked must now depend upon *both* arguments to the message: the graphical display doing the drawing, and the graphical object which is being drawn.

The `GraphicalDisplay` classes can each provide an implementation of the draw method, but these cannot depend on the types of the graphical object arguments: a complementary design could swap the methods' receiver and argument objects (so programmers would write `GraphicalObjects drawOn: Graphical Display`) this would allow different messages for each graphical object but not for different kinds of graphical displays. This problem is actually more common that it may seem in object-oriented designs. The visitor pattern, for example has a composite structure that accepts a visitor object embodying an algorithm to carry out over the composite (e.g. `Composite accept: aVisitor`): implementations of the accept method must depend upon the types of both composite and visitor [Gamma 1995].

Overloading in languages like Java or C++ can partially address this problem under certain circumstances. For example, Java allows methods to be distinguished based on the class of their arguments, so that a `ScreenDisplay` object can have different draw methods for displaying `Lines`, `Rectangles`, or `Arcs`:

```
abstract class GraphicalDisplay {
 public void draw(Line l) {
         // draw a line on some kind of display };
 public void draw(Rectangle r} {
         // draw a rectangle on some kind of display };
 public void draw(Arc a) {
         // draw an arc on some kind of display };
}
```

```
class ScreenDisplay extends GraphicalDisplay {
 public void draw(Line l) {
         // draw a line on a screen };
 public void draw(Rectangle r} {
         // draw a rectangle on a screen };
 public void draw(Arc a) {
         // draw an arc on a screen };
}
```

The problem here is that overriding is only resolved statically. Java will report an error in the following code:

```
Display d = new ScreenDisplay();
GraphicalObject g = new Line();
d.draw(g)
```

because the screen display class does not implement a `draw(GraphicalObject)` method.

The usual solution to this problem, in both Smalltalk and Java, is *double dispatch* [Ingalls 1986, Hebel 1990]: rather than implementing messages directly, method bodies send messages back to their arguments so that the correct final method body can depend on both classes. In this case, the `GraphicalDisplay` subclasses would each implement the draw methods differently, by asking their argument (the graphical object to be drawn) to draw themselves on a screen or on a printer:

```
ScreenDisplay>>draw: aGraphicalObject
         aGraphicalObject drawOnScreen: self

PrinterDisplay>>draw: aGraphicalObject
         aGraphicalObject drawOnPrinter: self
```

The key idea is that these methods encode the class of the receiver (`Screen` or `Printer`) into the name of the message that is sent. The `GraphicalObject` class can then implement these messages to actually draw:

```
Line>>drawOnScreen: aScreen
         "draw this line on aScreen"

Line>>drawOnPrinter: aPrinter
         "draw this line on aPrinter"
```

Each message send — that is, each dispatch — resolves the type of one argument. Statically overloaded implementations often generate “mangled” names for statically overloaded variants that similarly add type annotations to the names the virtual machine sees under the hood for compiled methods.

A few object-oriented languages, notably CLOS and Dylan [Bobrow 1998a, Keene 1989, Feinberg 1996], and various research extensions to Java [Boyland 1997, Clifton 2000] solve this design problem directly by supporting *multimethods*. A multimethod is simply a method that provides multiple dispatch, that is, the method body that is chosen can depend upon the type of more than one argument In this case code very similar to the Java code above could provide various different versions of the draw methods (one for each kind of `GraphicalObject`) within the `Display` classes, but the languages will choose the correct method to execute at runtime, based on the types of *all* the arguments in the message. The remainder of this paper describes how we implemented efficient multimethods as a seamless extension to Smalltalk.

3 Multimethods for Smalltalk

The first issue we faced in designing Smalltalk multimethods is that we wanted multimethods to fit in with the style or spirit of Smalltallk. Compared with most multimethod languages (especially CLOS) Smalltalk is lightweight, with a minimalist language design philosophy. A program is seen as a community of objects that communicate via message sends, and even "if" statements are technically implemented as messages to objects like true and false. An important aim of our design is that it should not change the basis of the language, and that multimethods should not affect Smalltalk programmers who choose not to write them.

The second issue is simply that Smalltalk, like Common Lisp, is a dynamically typed language, so that the language syntax does not, by default, include any specification of, or notation for, method types. As we've seen above, in many other object-oriented languages (such as Java and C++) method definitions must include type declarations for all their arguments even though the message sends will be dispatched in terms of just one distinguished argument.

Furthermore, in Smalltalk, programmers interact with programs on a per-method basis, using Smalltalk browsers. Source descriptions of these method objects are edited directly by programmers, and are compiled whenever methods are saved. Even when code is saved to files, these files are structured as "chunks" of code [Krasner 1983] that are written as sends to Smalltalk objects that can in turn, when read, reconstitute the code. Because of the way Smalltalk's browsers and files are set up, method bodies need not explicitly specify the class to which a method belongs. The class is implicitly given the context in which the message is defined.

Finally, Smalltalk provides reflective access to runtime *metaobjects* that represent the classes and methods of a running program, and allows a program to modify itself by manipulating these objects to declare new classes, change existing ones, compile or recompile methods, and so on. This arrangement is circular, rather than a simple layering, so that, for example, the browsers can be used to change the implementation of the metaobjects, even when those metaobjects will then be used to support the implementation of the browsers.

A language design to provide multimethods for Smalltalk must address all four of these issues: it must define how multimethods fit into Smalltalk's language model, it must provide a syntax programmers can use to define multimethods, browser support so that programmers can write those methods, and the metaobjects to allow programmers to inspect and manipulate multimethods. A key advantage of the Smalltalk architecture is that these three levels are not independent: the metaobjects can be used to support both the browsers and language syntax.

Design: Symmetric vs. Encapsulated Multimethods

There are two dominant designs for multimethods in object-oriented programming languages. Languages following CLOS or Dylan [Bobrow 1988, Feinberg 1996] provide *symmetric* multimethods, that is, where every argument of the multimethod is treated in the same way. One consequence of this is that multimethods cannot belong to particular classes (because object-oriented methods on classes treat the receiver (`self` or `this`) differently from all the other arguments. *Encapsulated* or *asymmetric* multimethods [Boyland 1997, Castagna 1995, Bruce 1995] are an alternative to symmetric multimethods: as the name implies, these messages belong to a class and are in some sense encapsulated within one class, generally the class of the receiver.

We consider that asymmetric multimethods are a better fit for Smalltalk than symmetric multimethods. Smalltalk's existing methods obviously rely on a single dispatch with a distinguished receiver object; its syntax and virtual machine support are all tied to that programming style. Similarly, Smalltalk being class-based can naturally attach encapsulated multimethods to a single class.

Syntax and Semantics

A Multimethod will differ from a singly dispatched method in two ways. First, *specializers* that describe the types for which the methods are applicable must be specified for their formal arguments. Second, it must be possible to provide multiple definitions (generally with different specializers) for a single message name. This is similar to the way in which Java allows a single method name to have multiple overloaded definitions with different argument types.

There are two ways this might be done in Smalltalk. The first is to change the parser and compiler to recognize a new *syntax* for multimethod specializers. The second is to allow method *objects* to be changed or converted programmatically using runtime messages, perhaps with browser support, such as pull-down specializer lists, or, with additional arguments to the metaobjects that create the method object itself. The first approach is based on the text-based, linguistic tradition of programming language design, while the second is based on a more modern, browser/builder approach that supplants the classical notion of syntax with the more contemporary approach of direct manipulation of first-class language objects.

While we used elements of both approaches to build our multimethods, we relied, in this case, primarily on the more traditional text-based approach of the sort taken by CLOS [Bobrow 1998a], Dylan [Feinberg 1996], and Cecil [Chambers 1992]. In CLOS, a type specializer is represented as a two element list:

```
(defmethod speak ((who animal))
(format t "I'm an animal: ~A~%" who))
```

Dylan, by contrast, uses :: to denote specialization:

```
define method main (argv0 :: <byte-string>, #rest noise)
          puts("Hello, World.\n");
end;
```

The angle brackets are part of the type name in Dylan. Dylan, as with other languages in the Lisp tradition, is permissive about the sorts of characters that may make up names. Cecil uses an @ sign to indicate that an argument is *constrained* (which is how they refer to their brand of specializers).

```
x@smallInt + y@smallInt
         { ^primAdd(x,y, {&errorCode | … })}
```

As a completely dynamically typed language, Smalltalk does not require type declarations for variables or method arguments. However, Smalltalk programmers have long used a type syntax using angle brackets, either before or after the qualified argument, even though such declarations have no effect on the execution of a program using them. The VisualWorks 2.5x Smalltalk compiler can recognize an "extended language" syntax in which method arguments are followed by angle-bracketed type specifiers. This trailing angle-bracketed type designation notation was first suggested for Smalltalk by Borning and Ingalls over twenty years ago [Borning 1982]. A similar syntax was used in Typed Smalltalk [Johnson 1988a], and is used in the Visual Works documentation as well as the Smalltalk Standard. These specifiers can contain Smalltalk literals, symbols, or expressions.

We have adopted this syntax to support multimethods. The necessary adaptation is quite simple, comparable with Boyland and Castagna's Parasitic Multimethods for Java [Boyland 1997]. To declare a multimethod, a programmer simply adds a class name within angle brackets after any method argument. This method body will then only be called when the message is sent with an argument that is (a subclass of) the declared argument type, that is via a multiple dispatch including any argument with a type specializer. Here is an example of this syntax for the Graphical Display problem:

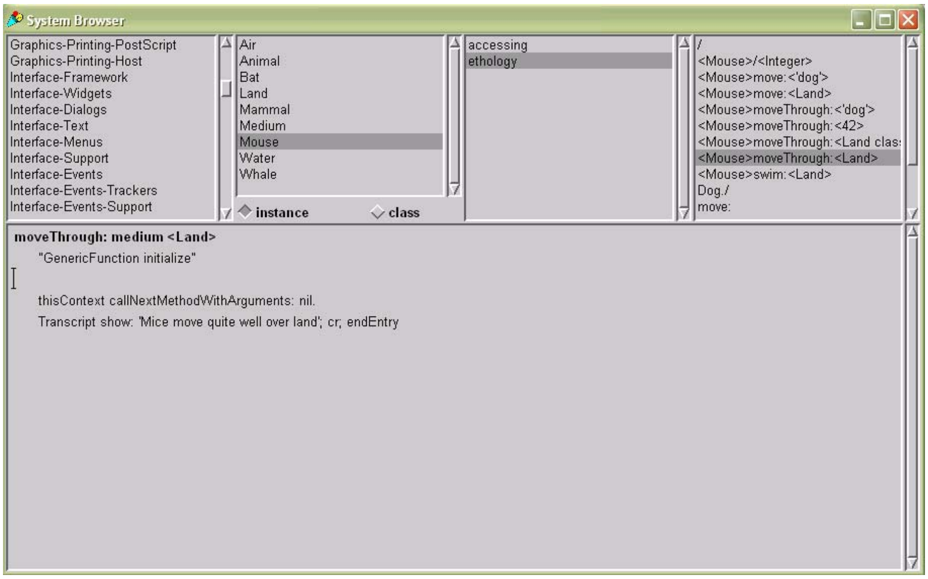
```
ScreenDisplay>>draw: aGraphicalObject <Line>
 "draw a line on a screen"

ScreenDisplay>>draw: aGraphicalObject <Arc>
 "draw an arc on a screen"
```

When a draw: message is sent to a `ScreenDisplay` object, the appropriate draw method body will now be invoked at runtime, with the decision of which message to invoke depending on the runtime classes of the object receiving the message, and any arguments with specializers. If no method matches, the message send will raise an exception, in the same way that Smalltalk raises a `doesNotUnderstand:` exception when an object receives a message it does not define.

These multimethods interoperate well with Smalltalk's standard methods and with inheritance, primarily because they are first sent (asymmetrically) to a receiver `(self)` object so their semantics are a direct extension of Smalltalk's standard method semantics. Multimethods may access instance and class variables based on their receiver, just as with standard Smalltalk methods. A multimethod defined in a subclass will be invoked for all arguments that match; otherwise an inherited method or multimethod in a superclass will be invoked. A multimethod can use a `super` send to invoke a standard method defined in a superclass, and vice versa. From this perspective, a "normal" Smalltalk method is treated exactly the same as a single multimethod body, where all arguments (other than the receiver) are specialized to `Object`. Our base multimethod design does not support one multimethod body delegating a message to another multimethod body defined in the same class, however, such common code can be refactored into a separate method and then called normally We have also experimented with a more flexible “call-next-method” scheme modeled after CLOS.

Browser Support

Languages that support multimethods have long been regarded as needing good programming environment support [Rosseau 1993]. Unlike most other programming languages, Smalltalk-80 has had an excellent integrated programming environment [Goldberg 1984] (and arguably has had one from before the start [Goldberg 1976]). Because this environment is itself written in Smalltalk we were able to exploit it to support Smalltalk multimethods.



Graphics-Printing-PostScript
Graphics-Printing-Host
Interface-Framework
Interface-Widgets
Interface-Dialogs
Interface-Text
Interface-Menus
Interface-Support
Interface-Events
Interface-Events-Trackers
Interface-Events-Support

Air
Animal
Bat
Land
Mammal
Medium
Mouse
Water
Whale

accessing
ethology

/
<Mouse>/<Integer>
<Mouse>move:<'dog'>
<Mouse>move:<Land>
<Mouse>moveThrough:<'dog'>
<Mouse>moveThrough:<42>
<Mouse>moveThrough:<Land clas
<Mouse>moveThrough:<Land>
<Mouse>swim:<Land>
Dog./
move:

```
moveThrough: medium <Land>
    "GenericFunction initialize"

    thisContext callNextMethodWithArguments: nil.
    Transcript show: 'Mice move quite well over land'; cr; endEntry
```

Fig. 1. A Smalltalk Browser displaying a multimethod

In fact, due to the design of the VisualWorks browsers, very few changes were required. For instance, while the Smalltalk `Parser` is selective about method selector syntax, the browsers are not. Any Smalltalk `Symbol` object (and perhaps other printable objects as well) can be used to index a method in the browsers. We exploited this fact to allow `MultiMethod` objects to appear with bracketed type specializers where their specialized arguments are to appear. Normal methods appear unchanged.

Metaobjects

Smalltalk is a computationally reflective language, that is to say, it is implemented in itself. The objects and classes that are used to implement Smallltalk are otherwise completely normal objects (although a few may be treated specially by the VM) but because they are used to implement other objects they are known as *metaobjects* or *metaclasses* respectively. Smalltalk programs are made up of metaobjects — Smalltalk methods are represented by instances of `Method` or `CompiledMethod` metaobjects, and Smalltalk classes by instances of `Class` metaobjects. The Smalltalk compiler (itself an instance of the `Compiler` class) basically translates Smalltalk language strings into constellations of these metaobjects. To implement multimethods in Smalltalk, we installed our own modified version of `Compiler` that understood the multimethod syntax and produced new or specialized metaobjects to implement multimethods.

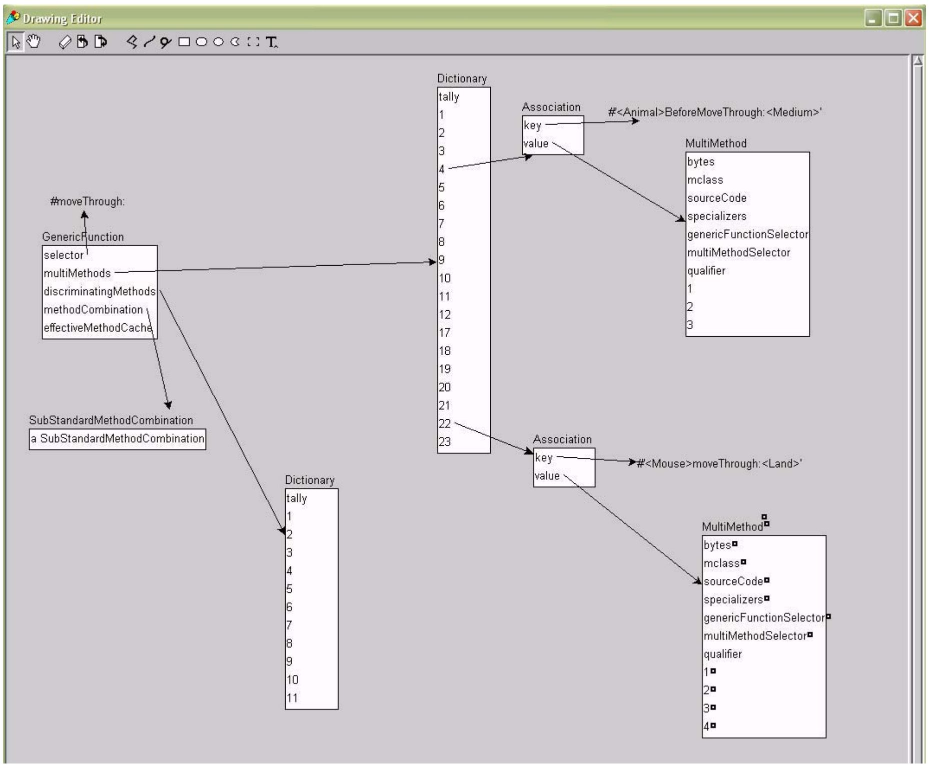


Fig. 2. A GenericMethod with its Multimethods

Our first implementation of multimethods was based on the CLOS MetaObject Protocol [Kiczales 1991]. It consists of the following metaobjects: `Multimethods`, `Specializers`, `GenericMessages`, `MethodCombinations`, and

DiscriminatingMethods. Fig. 2 shows the way these objects collaborate to represent multimethods.

A GenericMessage (GenericFunction in the figure) contains a Dictionary mapping specialized message selectors to their respective multimethod bodies. The GenericMessage's associated MethodCombination object orders these multimethod bodies to determine the correct method to invoke. A GenericMessage also contains a list of the DiscriminatingMethods that intercept method calls and start the multimethod dispatch. We describe each of these objects in turn below.

Multimethods

A MultiMethod metaobject represents one multimethod body. That is, it represents a method that can be dispatched with any or all of its arguments being taken into account, instead of just the first one. A multimethod must have one or more argument Specializers that determine the kinds of arguments to which the multimethod will respond. A multimethod can determine if is *applicable* to a series of arguments (via its specializers) and, if so, can run the code in its body when required.

Specializers

Specializers represent the argument to which a particular Multimethod applies. When a specializer is invoked, it determines if the argument passed to the multimethod matches that multimethod, or not. We currently use two different kinds of Specializers: ClassSpecializers, and EqualSpecializers. ClassSpecializers indicate that a multimethod applies when the corresponding argument is a member of the indicated class, or one of its subclasses. EqualSpecializers (which are modeled after CLOS's EQL specializers), match when an argument is equal to a particular object. Cecil [Chambers 1992], a prototype-based dynamic language with multimethods, gets EqualSpecializers for free, since all specializers are instances, not classes.

Generic Messages

A GenericMessage represents the set of all multimethods with the same name. (The name GenericMessage is derived by analogy with the similar Generic Function object in CLOS). When a GenericMessage is called, its job is to select, from among all its MultiMethods, only those that are consistent with the arguments it was called with (the *applicable* methods). These must also be sorted in the correct order, that is, from the most specific multimethod to the least specific multimethod.

Method Combinations

A MethodCombination object defines the order in which methods are called, and how different kinds of methods are treated. Again, our MethodCombinations are modeled after those in CLOS. Their job is to take the set of applicable methods that was determined by the GenericMessage to be "in-play" given the current arguments, and execute these in the manner that their qualifiers and the

MethodCombination itself prescribe. When a generic message finds more than one applicable method, these are sorted from most specific to least specific. This situation is analogous to a normal message send, where a call finds the most specific subclass's version of a method.

Multiple methods can apply because some will match more precisely, or specifically, at one or more argument sites. For example, consider the following two multimethods on a Stencil class (that draws multiple copies of an image along a path):

```
Stencil>>drawUsingShape: rectangle<Rectangle>
            OnDisplay: display <GraphicalDisplay>

Stencil>>drawUsingShape: shape<GraphicalObject>
            OnDisplay: display <ScreenDisplay>
```

Both of these multimethods would match a call where the first (shape) argument was a Rectangle and the second (display) argument a ScreenDisplay. In this case, the MethodCombination will sort these in the order shown, because the specifier on the first multimethods's first argument is more specific that the specifier on the first argument of the second multimethod.

MethodCombination objects can be thought of as examples of the Strategy design pattern [Gamma 1995]. MethodCombination objects represent the rules for combining and calling a multimethod's bodies. A GenericMessage can change the way that its methods are dispatched by designating a new MethodCombination object. Of course, the multimethods themselves must be written carefully in order to allow changes in the combination scheme to make sense. That is to say, methods are normally written without having to concern themselves with the possibility of being combined in exotic, unexpected ways.

Discriminating Methods

Any message send in a Smalltalk program needs to be able to invoke a multimethod. Whether a multimethod or a “normal” Smalltalk method will be invoked depends only upon whether any multimethod bodies (i.e. any methods with specializers) have been defined for that message name. That is (as with normal Smalltalk methods) the implementation of the method is solely the preserve of the receiver of the message (or, from another perspective, the classes implementing that method). This design has the short-term advantage that no performance overhead will be introduced in Smalltalk programs that do not use multimethods, or for sends of “normal” methods in programs that also include multimethods; and the longer-term advantage that classes can turn their methods into multimethods (by adding specialized versions), or vice versa, without any concern for the clients of those classes, or the callers of those messages.

In practice, this design means that our multimethod dispatch must intercept the performs this interception. Smalltalk cannot intercept an incoming message until one dispatch, on the first argument, has already been done. A DiscriminatingMethod (again named after the analogous discriminating functions in CLOS) is a MethodWrapper [Brant 1998] that acts as a decorator around the standard Smalltalk CompiledMethod object.

Fig. 3 shows how the multimethods and `DiscriminatingMethods` hook into standard Smalltalk classes. All standard Smalltalk class metaobjects (including, in this figure, the `Mouse` class) contain a `MethodDictionary` implemented as two parallel arrays. The first array contains method selectors. For our multimethods, as well as the standard selectors (`#moveThrough` in the example in this figure) we include specialized selectors (`#<Mouse>moveThrough:<Land>`). The second array normally contains method bodies: in our implementation the standard selector (`#moveThrough` that will actually be sent by program code) maps to a `DiscriminatingMethod` that will invoke the multimethod dispatch, while the specialized selector maps to the object representing the `MultiMethod` body. Because this method includes two specializers (`<Mouse>` and `<Land>`) it is linked to two `ClassSpecializer` objects.

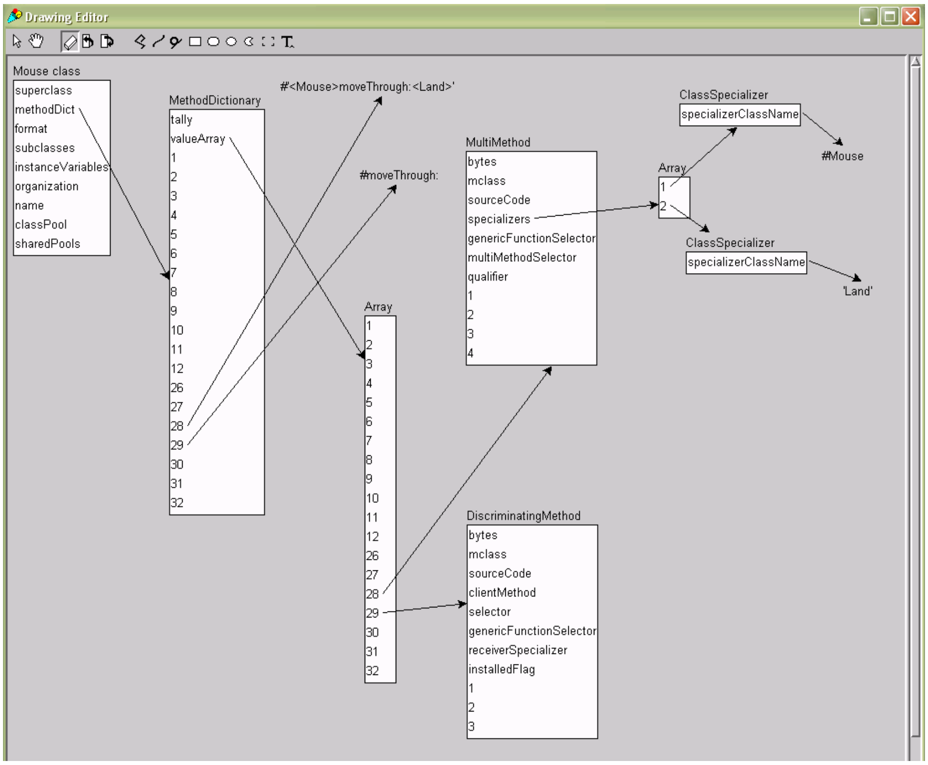


Fig. 3. A Class's MethodDictionary mapping a DiscriminatingMethod and Multimethods

The relationship between `MultiMethods`, `GenericMessages`, and `DiscriminatingMethods` is as follows. There is one `GenericMessage` for every message name in the system that has at least one specialized method body. Every method body defined with a specializer is represented by a `MultiMethod` object (and its associated `Specializers`) — all of these are known by their

GenericMessage, and can be chosen by its MethodCombination. Finally, every class that can understand this multimethod name (i.e. that has at least one MultiMethod body defined) will have a DiscriminatingMethod stored under that name that is again linked to its GenericMessage.

Invoking Multimethods
All these metaobjects collaborate to implement the dispatch whenever a program sends a message to an object that implements that message using multimethods. First, a DiscriminatingMethod is used to gain control. The DiscriminatingMethod then forwards the message send and its argument values to the GenericMessage object for all multimethods with that name. Next, the GenericMessage iterates across each candidate MultiMethod looking for all the applicable MultiMethods, that is, all MultiMethods whose Specializers match the actual arguments of the message send.

The GenericMessage then sorts the applicable methods in order of applicability, and passes the list to the GenericMessage's MethodCombination object. The MethodCombination then selects and executes the body of the chosen MultiMethod. This result is then returned (via the GenericMessage) to the DiscriminatingMethod, and thus is returned (as normal) to the caller of the multimethod.

As with CLOS, these objects are designed to allow the caching of partial results.

4 Examples

In this section, we present some examples to show how multimethods could be used to support the design of Smallltalk programs.

Eliminating Class Tests: Smalltalk methods often use explicit tests on the classes of their arguments. For example, the method to add a visual component to a component part in the VisualWorks interface framework behaves differently if the argument is a BorderedWrapper. This is implemented using an explicit class test:

```
ComponentPart>>
  add: aVisualComponent borderedIn: aLayoutObject

^(aVisualComponent isKindOf: BorderedWrapper)
  ifTrue: [aVisualComponent layout: aLayoutObject.
      self addWrapper: aVisualComponent]
  ifFalse: [self addWrapper:
   (self borderedWrapperClass on: aVisualComponent
              in: aLayoutObject)]
```

With multimethods, this could be refactored to two multimethods, one handling BorderWrapper arguments, and another the rest:

```
ComponentPart>>
  add: aVisualComponent <BorderWrapper>
  borderedIn: aLayoutObject
```

```
    aVisualComponent layout: aLayoutObject.
    ^self addWrapper: aVisualComponent.

  ComponentPart>>
    add: aVisualComponent <Object>
    borderedIn: aLayoutObject

  ^self addWrapper:
    (self borderedWrapperClass on: aVisualComponent
                 in: aLayoutObject)
```

Visitor: The following example, drawn from [Brant 1998], illustrates the impact of multimethods on the Visitor pattern [Gamma 1995]. First, consider a typical Smalltalk implementation of Visitor:

```
   ParseNode>>acceptVistor: aVisitor
      ^self subclassResponsibility

   VariableNode>>acceptVistor: aVisitor
      ^aVisitor visitWithVariableNode: self

   ConstantNode>>acceptVistor: aVisitor
      ^aVisitor visitWithConstantNode: self

   OptimizingVisitor>>visitWithConstantNode: aNode
      ^aNode value optimized

   OptimizingVisitor>>visitWithVariableNode: aNode
      ^aNode lookupIn: self symbolTable
```

When `MultiMethods` are available, however, the double-dispatching methods in the `ParseNodes` disappear, since the type information does not need to be hand-encoded in the selectors of the calls to the `Visitor` objects. Instead, the `Visitor` correctly dispatches sends of `visitWithNode` to the correct `MultiMethod`. Thus, adding a `Visitor` no longer requires changing the `ParseNode` classes.

```
  OptimizingVisitor>>visitWithNode: aNode <ConstantNode>
           ^self value optimized

  OptimizingVisitor>>
           visitWithNode: aNode <VariableNode>
           ^aNode lookupIn: self symbolTable
```

5 Implementation

We have experimented with a number of different implementations for multimethods. The first and simplest scheme is just to execute the Smalltalk code for the metaobjects directly. While acceptable for simple examples, such a strategy proved unacceptably slow, and so we therefore experimented with a number of different optimizations, some of which can execute code using multimethod as quickly as handwritten Smalltalk code for multiple dispatching. This section describes these implementations, and then presents our performance results.

Metaobjects: Our initial, unoptimized implementation simply executed the Smalltalk code in the metaobjects to dispatch multimethods: A method wrapper is used to gain control (adding about an order of magnitude to the dispatch process), the generic message iterates across each multimethod body and its specialzers, the resulting list is sorted, and so on. Even before performance testing, it seemed obvious that this approach would be too slow to be practical. Fortunately, there is quite a bit that can be done to speed things up.

Dictionary: Our first optimization uses a Smalltalk Dictionary to map from arrays of specializers to target methods. It is, in effect, a simple implementation of the hashtable scheme discussed by Kiczales and des Rivieres in [Kiczales 1991]. Our scheme relies on the fact that it would be applied in a `DiscriminatingMethod`, and left out the first argument: the other argument classes are cached in a table so that the applicable multimethod body can be found directly.

Case: Our second optimization was to directly test the classes of each argument and calls the appropriate method. The idea is that the decision tree itself is inlined as in a case statement. We wrote this version by hand, but code to implement these case tree dispatchers could be synthesized automatically, should this approach prove practical.

Multidispatch: Our third optimization is a generalization of the double dispatch scheme described by Ingalls [Ingalls 1986]. Instead of merely redispatching once, redispatchers are generated so that each argument gets a chance to dispatch. Hence, triple dispatch is performed for three argument multimethods, quadruple dispatch for four, octuple dispatch for eight, etc. At each step, identified class/type information is “mangled” into the selectors, that is, we automatically generate the same code that a programmer would write to implement multiple dispatch in Smalltalk . Since this approach takes advantage of the highly optimized dispatching code in the Visual Works Virtual Machine, we expected its performance to be quite good. The main problem with multiway dispatch is that a large number of methods may be generated:

$$\begin{aligned}|D| = {} & |S_1| \\ & + |S_2| \times |S_1| \\ & + |S_3| \times |S_2| \times |S_1| \ldots \\ & + |S_n| \times \ldots \times |S_2| \times |S_1|\end{aligned}$$

or, alternately,

$$|D| = \sum_{i=1}^{n} \prod_{j=1}^{i} |S_j|$$

where $|S_j|$ denotes the cardinality of the set of specializers for the indicated argument of a particular generic message and $|D|$ is the cardinality of the set of dispatching methods which must be generated.

Our multidispatch code generates all the required multidispatch methods automatically. They are all placed in a special Smalltalk protocol category: 'multidispatch methods'. These methods are named using the compound selector syntax inherited from VisualWorks 2.0 that has its roots in the Borning and Ingalls multiple inheritance design [Borning & Ingalls 1982]. These selectors allow periods to be included as part of the message selector name. The original selector is placed at the beginning of each multidispatch selector, with dots replacing the colons. Argument specifiers are indicated using 'arg1', followed by the specializer for the argument, if one was recognized by a previous multidispatch method.

Our implementation generates an additional, final dispatch to the initial multimethod receiver so that the target multimethod body can be executed as written. In one sense, this final dispatch is a concession to the low-level asymmetry inherent in our Smalltalk implementation. In effect, this final ricochet closes the multidispatched circle. This introduces an additional factor of two into the last term in the formula above. Given this, the number of methods we generate becomes:

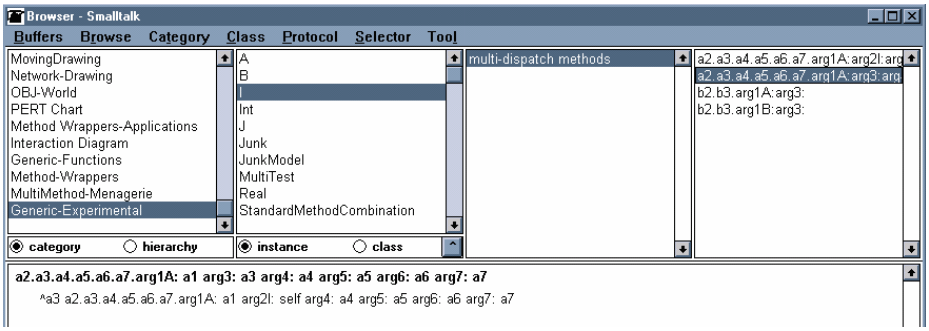
$$\begin{aligned}|D| = {} & |S_1| \\ & + |S_2| \times |S_1| \\ & + |S_3| \times |S_2| \times |S_1| \ldots \\ & + |S_n| \times \ldots \times |S_2| \times |S_1| \times 2\end{aligned}$$

Note that the class of the recipient of this final dispatch will be pre-determined by the time this call is made, hence, virtual machines that employ inline caching mechanisms will incur minimal overhead for all but the initial call to such methods.

We can reduce the number of methods we have to generate by shuffling the order in which the arguments are dispatched. Since each S_j introduced is a factor in every subsequent term of the formula above, dispatching from the lowest cardinality specializer up to the highest will minimize the number generated methods. Of course, Smalltalk forces us to start with the first argument, S_1, instead of whichever we wish. Since leftmost factors are repeated more often, reordering multiway dispatch so that the smaller factors are the ones that recur minimizes the number of methods that must be generated.

Two additional optimizations are possible. Were the target method's body merged with the final set of dispatching methods, the final "× 2" factor in the final term of the equation above could be elided. Also, only arguments that are actually *specialized* need be redispatched. That is to say, if the cardinality of the set of specializers is one (that is, the argument is <u>not</u> specialized), then it can be bypassed. To put it another way, <u>only</u> arguments for which more than one specializer is present need be treated as members of the set of specializers.

While the formula above might suggest to some readers that in the worst case there is cause for concern that this generalized multiway dispatch scheme might entail the generation of an unacceptably large number of methods, we believe that the potential for practical problems with this approach is rather low. Indeed, Kiczales and Rodriguez [Kiczales 1990] observed that only four percent of the generic functions in a typical CLOS application specialized more than a single argument. We expect that multimethods with large numbers of specializers on multiple arguments to be rare birds indeed.



```
MovingDrawing
Network-Drawing
OBJ-World
PERT Chart
Method Wrappers-Applications
Interaction Diagram
Generic-Functions
Method-Wrappers
MultiMethod-Menagerie
Generic-Experimental
```

```
A
B
I
Int
J
Junk
JunkModel
MultiTest
Real
StandardMethodCombination
```

```
multi-dispatch methods
```

```
a2.a3.a4.a5.a6.a7.arg1A:arg2I:arg
a2.a3.a4.a5.a6.a7.arg1A:arg3:arg
b2.b3.arg1A:arg3:
b2.b3.arg1B:arg3:
```

```
a2.a3.a4.a5.a6.a7.arg1A: a1 arg3: a3 arg4: a4 arg5: a5 arg6: a6 arg7: a7
    ^a3 a2.a3.a4.a5.a6.a7.arg1A: a1 arg2I: self arg4: a4 arg5: a5 arg6: a6 arg7: a7
```

Fig. 4. Generated Multidispatch Example

Performance

Table 1 compares the performance of the various implementation techniques with the cost of performing an Ingalls-style double dispatch. This is shown in **row 1**. The cost for the simplest standard Smalltalk single-dispatched method call that returns the called object is shown in **row 2.** Returning self in Smalltalk is a special case both in the bytecode and the compiler, however it is only five times faster than a double dispatch that must do significantly more work. Each row in the table shows the results of a single implementation (the number in parenthesis in the leftmost column is the number of arguments the multimethod dispatched upon). We make multiple runs of each benchmark, timing 1,000,000 (multi)method sends, and report the minimum and maximum invocation times for each run.

Row 3 shows the performance of the straight-ahead Smalltalk implementation of multimethods, giving the overhead when a target method dispatching on two method arguments simply returns itself. That is, we take about 600 microseconds to do nothing. **Row 4**, with full method combination support calling an overridden multimethod is extremely slow. This sort of dismal performance is not unheard of when reflective facilities are used. For instance [Palsberg 1998] found 300:1 performance decreases for their reflective implementations of variants on the Visitor pattern. **Row 5** gives the performance of a simple extension to this scheme, where the final multimethod body lookup is cached, giving a fivefold increase in performance but still being slow relative to a hand coded implementation.

Rows 6 and **7** give the performance of the Dictionary and case-statement lookups respectively, dispatching on three specialized arguments. Again, these optimizations provide another order of magnitude but are still twenty times as slow as the basic multiple dispatch.

Rows 8 and **9** finally show the performance of the generated multidispatch implementation, **row 8** again dispatching on three arguments and **row 9** on seven. Here at last is an implementation that performs at roughly the same speed as the standard Smalltalk system, because our generated code is effectively the same as the code an experienced Smalltalk programmer would write to implement multiple dispatch. This is the implementation we have adopted in our system.

Table 1. Performance Results

Dispatch Type	nanosec min	nanosec. max	Ratio
1. Multidispatch (2 args)	521	524	1.00
2. Tare (^self) (1 arg)	90	120	0.20
3. Metaobjects (^self) (2 args)	597,000	624,000	1168
4. Metaobjects (super) (2 args)	679,000	750,000	1367
5. Metaobjects cached (2 args)	117,000	125,000	231
6. Dictionary (3 args)	13227	13335	25
7. Case (inline) (3 args)	10654	10764	20
8. Multidispatch (3 args)	633	779	1.35
9. Multidispatch (7 args)	1200	1221	2.32

200MHz Pentium Pro
1,000,000 calls/multiple runs

There are a variety of trade-offs that must be considered among these approaches. The "pure" Smalltalk solution is relatively easy to use, but performs so poorly that it is little more than a toy. It is a testament to the power of reflection that the range of strategies for improving this performance can be addressed at all from within the Smalltalk programming environment itself. Still, these are not without their costs. The multidispatch approaches can litter the method dictionaries with dispatching methods. These, in turn, beg for improved browsing attention.

The final performance frontier is the virtual machine itself. While possible, this would require a way of controlling the dispatch process "up-front" [Foote 1989], and would greatly reduce portability. Given that performance of our multidispatch scheme is as quick as standard Smalltalk, we consider that the complexity of changing the virtual machine is not justified by the potential increase in dispatching performance.

6 Discussion

In this section we address a number of issues regarding the provision of multimethods in Smalltalk.

Access to Variables: although multimethods are dispatched on multiple arguments, they remain encapsulated within a single class as in standard Smalltalk and can only access instance variables belonging to `self`. It is, of course, possible to generate accessor methods so that multimethods could access instance variables belonging to all arguments. This is, in essence, the approach take by CLOS. Given that we aimed to retain as much of Smalltalk's object model as possible (and Smalltalk's strong variable encapsulation is an important part of that model) we elected not to change this part of the language. As in standard Smalltalk, programmers can always choose to provide instance variable accessor methods if they are needed by particular multimethods. Indeed, such accessors, together with a judicious choice of `MethodCombination` objects, allow multimethods to be programmed in a symmetric style.

Class-Based Dispatch: as in standard Smalltalk, our multimethods are dispatched primarily based upon the classes of arguments. Smalltalk has an *implicit* notion of object type (or *protocol*), based on the messages implemented by a class, so two classes can implement the same interface even if they are completely unrelated by inheritance. We considered providing specializers that would somehow select methods based on an argument's *interface* or *signature*, but this would require an *explicit* notion of an object's type signature, which standard Smalltalk does not support (although extensions to do so have long been proposed [Borning 1987, Lalonde 1986]). One advantage of class-based dispatch, given that Smalltalk supports only single inheritance, is that class-based selectors will never be ambiguous, as is possible with multiple inheritance or multiple interfaces.

Our implementation does support instance-based `EqualSpecializers` as well. We have not as yet made a detailed assessment of either their impact on performance, or of their overall utility.

Portability and Compatibility: we have taken care to maximize the portability of our multimethod design across different Smalltalk implementations. Our syntax is designed so that it is completely backwards compatible with existing Smalltalk syntax and to impose no overhead on programmers if multimethods are not used. Similarly, our design requires no changes to Smalltalk virtual machines and adds no performance penalty if multimethods are unused. The largest portability difficulties are with individual Smalltalk compilers and browsers, as these differ the most between different language implementations. A final aspect of portability relates to the compiled code for optimized implementations. Because the generated multidispatch code does not depend on any other part of the system, it can be compatible with Smalltalk systems without the remainder of the multimethod system.

Method Qualifiers — Extending Method Combinations: A great advantage of building multimethods by extending Smalltalk's existing metaobjects is that our implementation can itself be extended by specializing our metaobjects. We have implemented a range of extended method combination schemes, modeling those of CLOS and Aspect/J, to illustrate this extensibility.

Our extended method combination scheme allows `MultiMethods` to be given `Qualifiers`. `Qualifiers` are symbols such as `#Before`, `#After`, or `#Around`. These qualifiers indicate to `MethodCombination` objects the role these methods are to play, and how they should be executed. There are no a priori limitations on these qualifiers; they can be any symbols that the `MethodCombination` objects can recognize. As in CLOS and Aspect/J, we provide some stock `MethodCombination` objects that implement before, after, and around methods that execute before, after, or before-and-after other methods in response to a single message send [Brant 1998, Kiczales 2001]. We also provide a `MethodCombination` that emulates the Beta [Kristensen 1990] convention of executing methods from innermost to outermost. We also provide a `SimpleMethodCombination` object that executes its applicable method list in the order in which it is passed to the `MethodCombination` object.

These extended method combination metaobjects interpret the qualifiers during multimethod dispatch. The main change is that more multimethod bodies can match a particular method send, because a qualified method can execute in addition to other methods that also match the arguments of a message send. For example, as in CLOS, all before (or after) multimethods will execute before (or after) the one unqualified message chosen by the base multimethod dispatch.

Our current implementation of extended method combination is experimental. In particular, qualifiers must be assigned programmatically to multimethods as we have not yet provided syntactic or browser support.

Our design, as well as the designs of Smalltalk and CLOS, for that matter, is distinguished from more recent work based upon Java derivatives [Boyand 1977, Kiczales 2001] in that given that each is built out of objects, programmers can extend these objects themselves to construct any mechanism they want. It is a testimony to the designers of Smalltalk and CLOS [Gabriel 1991, Bobrow 1993] that principled architectural extensions, rather than inflexible, immutable preprocessor artifice, can be employed to achieve this flexibility.

Language design, it has been said, is not about what you put in, but about what you leave out. A system built of simple, extensible building blocks allows the designer to evade such painful triage decisions. The real lesson to be gleaned from the metalevel architectures of Smalltalk and CLOS is that if you provide a solid set of building blocks, programmers can construct the features they really need themselves, their way. This might be thought of as an application of the “end-to-end principle” [Saltzer 1981] to programming language design.

Programming Languages Versus Idioms and Patterns: Finally, our work raises the philosophical question of when programming idioms or design patterns should be incorporated into programming languages. From a pragmatic perspective, it is unnecessary to add multimethods into Smalltalk because multiple dispatch can be programmed quite effectively using idioms such as double dispatch [Ingalls 1986] or the Visitor pattern [Gamma 1995]. Our most efficient implementation merely matches the performance of these hand-coded idioms — some of our more basic implementations perform significantly worse — so efficiency is not a reason for adopting this extension.

Indeed, our harmonious melding of the CLOS MOP atop Smalltalk’s kernel objects (the Smalltalk “MOP”, if you will) suggests that neither single dispatch nor multi-dispatch is more fundamental that the other. Instead, they can be seen as complements or duals of each other. Single dispatch can be seen as merely a predominant, albeit prosaic special case of generalize multiway dispatch. Alternately, our results show that you can curry your way to multiple dispatch in any polymorphic, single dispatch language, one argument at a time.

In general, we consider that an idiom — such as double dispatch — should be incorporated into a language when it becomes very widely used, when a hand coded implementation is hard to write or to modify, when it can be implemented routinely, and at least as efficiently as handwritten code. The Composite and Proxy patterns, for example, may be widely used, but their implementations vary greatly, while implementing the Template Method pattern is so straightforward that it requires no additional support. On the other hand, the Iterator pattern is also widely used, but its implementations are amenable to standardization, and so we find Iterators incorporated into CLU, and now Java 1.5 and C#.

Multimethods are particularly valuable as Mediators. Since, for instance, a binary multimethod can be seen as belonging to either both or neither of a pair of class it specializes, it can contain glue that ties them together, while leaving each of its specializing classes untouched. The promise of clean separation of concerns, however admirable, is honored, alas, in many systems mainly in the breach [Foote 2000]. Multimethods are ideal in cases where mutual concerns arise among design elements that had heretofore been cleanly separated. Multimethods can help when concerns converge.

We believe that multiple dispatch is sufficiently often used; sufficiently routine; sufficiently arduous to hand code; and that our (and others) implementations are sufficiently efficient for it to be worthwhile to include into object-oriented programming languages.

7 Related Work

Multiple dispatch in dynamic languages was first supported in the LISP based object-oriented systems LOOPS and NewFlavours [Bobrow 1983; 1986]. As an amalgam of these systems, the Common Lisp Object System incorporated and popularized multiple dispatch based on generic functions [Bobrow 1988a , Keene 1989]. CLOS also incorporated a range of method combinations, although more recently these have also been adopted by aspect-oriented languages, particularly Aspect/J [Kiczales 2001]. Dan Ingalls described the now standard double-dispatch idiom in Smalltalk in what must be the OOPSLA paper with the all-time highest possible power-to-weight ratio [Ingalls 1986]. All these systems had the great advantage of dynamic typing, so were able to avoid many of the issues that arise in statically typed languages.

The first statically typed programming language with object-oriented multiple dispatch was the functional language Kea [Mugridge 1991]. While a range of statically typed languages provide overloading, (Ada, C++, Haskell, Java) satisfactory designs for incorporating dynamically dispatched multimethods into statically typed languages proved rather more difficult to develop. Craig Chamber's Cecil language [Chambers 1992] provided a model where multimethods were encapsulated within multiple classes to the extent that the multimethods were specialized on those classes. Further developments of Cecil demonstrated that statically typed multimethods could be integrated into practical languages and module systems with separate compilation. Cecil-style multimethods have also been incorporated into Java [Clifton 2000], and have the advantages of a solid formal foundation [Bruce 1995, Castagna 1995]. Bjorn Freeman-Benson has also proposed extending Self with Multimethods [Chambers 1992]. Rather than providing multiple dispatch by extending message sends Leavens and Millstein have proposed extending Java to dispatch on tuples of objects [Leavens 1998].

Closer to the design in this paper are Boyland and Castangna's Parasitic Multimethods. These provide a type-safe, modular extension to Java by dispatching certain methods (marked with a 'parasitic' modifier) according to the types of all their arguments [Boyland 1997]. As with our system, the parasitic design treats multimethods differently from normal ("host") messages, and then the distinguished receiver argument differently from the other arguments of a message. Multimethods are contained within their receiver's class and may access only those variables that are members of that class. Boyland and Castagna note that much of the complexity of their system comes from their goal of not changing Java's existing overloading rules, and recommend that future languages support only dynamic dispatch — ironically perhaps, the resulting language would be quite similar in expressiveness to Smalltalk with Multimethods.

The Visitor pattern is one of the main contexts within which double-dispatch is generally applied [Gamma 1995]; as with many patterns, Visitor has spawned a mini-industry of research on efficient implementation [Palsberg 1998, Grothoff 2003] that sometimes go as far as raising the specter of a Visitor-oriented programming "paradigm" [Palsberg 2004]. Similarly, incorporating features from Beta into more mainline (or at least less syntactically eccentric) object-oriented languages has also been of interest of late, with most work focusing on the Beta type system [Thorup 1997] although "inner-style" method combination has recently been adapted to a Java-like language design [Goldberg 2004].

Our work also draws on a long history of language experimentation, particular in Smalltalk. The dot-notation for extended selectors was originally proposed for multiple inheritance [Borning 1982] but has been used to navigate part hierarchies [Blake 1987]. More recent work on Array-based programming [Mougin 2003] employs somewhat similar techniques to extend Smalltalk, although without providing an extensible meta-model. Scharli et al. [2004] describe a composable encapsulation scheme for Smalltalk that is implemented using method interception techniques. This encapsulation model could be extended relatively straightforwardly to our multimethods, and would have the advantage that multimethods could thereby access private features of their argument objects.

This work draws upon many techniques developed over many years for meta-level programming [Smith 1983, Maes 1987a,b], both in Smalltalk and other languages. Our `DiscriminatingMethods` are derivations of `MethodWrappers` [Brant 1998], and the notion of extending method dispatch by meta-level means goes back at leat to CLOS and LOOPS [Bobrow 1983, Kiczales 1991]. Coda [McAffer 1995] provides an extended Smalltalk meta-object system that has been used to distribute applications across large scale multiprocessors. MetaclassTalk provides a more complete CLOS-style metaobject system for Smalltalk, again implemented with MethodWrappers, that has been used to implement various aspect-oriented programming constructs [Rivard 1997].

Finally, Bracha and Ungar [Bracha 2004] have classified the features of reflective systems into *introspection* (self-examination of a program's own structure); *self-modification* (self explanatory); *executing* dynamically generated code (ditto); and *intercession* (self-modification of a language's semantics from within). According to their taxonomy, Smalltalk scores highly on all categories except intercession. The dispatching metalevel we present in this paper can be seen either as a strong argument that Smalltalk does, in fact, provide powerful intercession facilities, or, more humbly, that straightforward, portable extensions can add these facilities to Smalltalk.

8 Conclusion

Though Smalltalk does not support multimethods, they can be built by programmers who understand Smalltalk's reflective facilities. There are several ways to go about this, and they differ dramatically in terms of power and efficiency. Taken together, they demonstrate the power of building programming languages out of objects, and opening these objects to programmers, and teach some interesting lessons.

One is that syntax matters. To build multimethods, we needed to be able to modify the compiler to support argument specializers.

A second lesson, however, is that when programs are objects, there are other mechanisms besides syntax that an environment can use to change a program. Our browsers support multimethods because methods have a uniform interface to these tools that allows our multimethod syntax to be readily displayed. Furthermore, since multimethods are objects, their attributes are subject to direct manipulation by these tools.

A third lesson is that runtime changes to objects that define how an object is executed are an extremely powerful lever. Using method wrappers to change the way methods act on-the-fly provides dramatic evidence of this.

A fourth is that there is a place for synthesized code, or code written by programs rather than programmers, in reflective systems. Generative programming [Czarnecki 2000] approaches have their place. Our efficient multiway dispatch code made use of this, allowing our reflective implementation to perform as well as hand-written code, without any changes to the Smalltalk virtual machine.

Our experience makes a powerful case for building languages out of objects. By doing so, we allow to the very objects from which programs are made be a vehicle for the language's own evolution, rather than an obstacle to it, as is too often the case.

To conclude, we have designed and implemented efficient multimethod support for Smalltalk. Our multimethods provide a very clean solution: programmers can define them using a simple extended syntax, their semantics are quite straightforward, they interoperate well with Smalltalk’s metaobjects, they impose no syntactic or runtime overhead when they are not used, and they are as efficient to execute as comparable hand-written code using sequences of single dispatches.

References

[Benoit 1986] Ch. Benoit, Yves Caseau, Ch. Pherivong, Knowledge Representation and Communication Mechanisms in Lore. *ECAI 1986*, 215-224

[Blake & Cook 1987] D. Blake and S. Cook. On Including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk. In *ECOOP Proceedings* 1988, 41-50.

[Bobrow 1983] Daniel G. Bobrow. *The LOOPS Manual.* Xerox Parc, 1983

[Bobrow 1986] Bobrow, D.G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., and Zdybel, F. CommonLoops: Merging Lisp and Object-Oriented Programming. In *OOPSLA Proceedings*.1986.

[Bobrow 1988a] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System Specification X3J13. *SIGPLAN Notices*, Volume 23, September 1988

[Bobrow 1988b] Daniel G. Bobrow and Gregor Kiczales. The Common Lisp Object System Metaobject Kernel -- A Status Report. In Proceedings of the 1988 Conference on Lisp and Functional Programming, 1988.

[Bobrow 1993] Daniel G. Bobrow, Richard P. Gabriel, Jon L. White, CLOS in Context: The Shape of the Design Space, in Object-Oriented Programming: The CLOS Perspective, Andreas Paepcke, editor, MIT Press, 1993, http://www.dreamsongs.com/NewFiles/clos-book.pdf

[Borning & O'Shea, 1987] Alan Borning and Tim O'Shea. Deltatalk: An Empirically and Aesthetically Motivated Simplification of the Smalltalk-80 Language. In ECOOP Proceedings,1987, 3-12.

[Borning & Ingalls 1982] A. H. Borning and D. H. H. Ingalls. A Type Declaration and Inference System for Smalltalk. In *POPL Proceedings*, 1982, 133-141.

[Boyland & Castagna 1997] John Boyland and Giuseppe Castagna Parasitic Methods: An Implementation of Multi Methods for Java. In *OOPSLA Proceedings* 1997.

[Bracha 2004] Gilad Bracha, David Ungar: Mirrors: design principles for meta-level facilities of object-oriented programming languages. In OOPSLA Proceedings, 2004. 331-344

[Brant 1998] John Brant, Brian Foote, Don Roberts and Ralph Johnson. Wrappers to the Rescue. In *ECOOP Proceedings*, 1998.

[Bruce 1995] Kim Bruce , Luca Cardelli , Giuseppe Castagna , Gary T. Leavens , Benjamin Pierce, On binary methods*, Theory and Practice of Object Systems*, v.1 n.3, p.221-242, Fall 1995

[Caseau 1986] Yves Caseau, An Overview of Lore. *IEEE Software* 3(1): 72-73

[Caseau 1989] Yves Caseau, A Model for a Reflective Object-Oriented Language, SIGPLAN Notices 24(4), 22-24

[Castagna 1995] Giuseppe Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431--447, May 1995

[Chambers 1992] Craig Chambers. Object-Oriented Multimethods in Cecil. In *ECOOP* Proceedings, 1992

[Clifton 2000] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for java. In *Proceedings of OOPSLA 2000*, 130-145.

[Czarnecki 2000] K. Czarnecki, U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000

[Deutsch 1984] L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. *In Proceedings of the Tenth Annual ACM Symposiumon Principles of Programming Languages,* 1983, 297-302

[Feinberg 1996] Neal Feinberg, Sonya E. Keene, Robert O. Mathews, and P. Tucker Washington. *The Dylan Programming Book.* Addison-Wesley Longman, 1996

[Foote & Johnson 1989] Brian Foote and Ralph E. Johnson. Reflective Facilities in Smalltalk-80. In *OOPSLA '89 Proceedings,* 1989, 327-335

[Foote & Yoder 1998] Metadata. In Proceedings of the *Fifith Conference on Pattern Languages of Programs (PLoP '98)* Monticello, Illinois, August 1998. Technical Report #WUCS-98025 (PLoP '98/EuroPLoP '98) Dept. of Computer Science, Washington University September 1998

[Foote 2000] Brian Foote and Joseph W. Yoder, Big Ball of Mud, in *Patterns Languages of Program Design 4* (PLoPD4), Neil Harrison, et al., Addison-Wesley, 2000

[Gabriel 1991] Richard P. Gabriel, Jon L. White, Daniel G. Bobrow, CLOS: Integrating Object-Oriented and Functional Programming, *Communications of the ACM*, Volume 34, 1991

[Gamma 1995] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software, Addision-Wesley, 1995.

[Grothoff 2003] C. Grothoff. Walkabout revisited: The runabout. In *ECOOP Proceedings*, 2003.

[Goldberg 1976] Adele Goldberg and Alan Kay, editors, with the Learning Research Group. *Smalltalk-72 Instruction Manual.* Xerox Palo Alto Research Center

[Goldberg 1983] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, MA, 1983

[Goldberg 1984] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, Reading, MA, 1984

[Goldberg 2004] David S. Goldberg, Robert Bruce Findler, Matthew Flatt. Super and inner: together at last! In *OOPSLA Proceedings* 2004, 116-129

[Hebel 1990] Kurt J. Hebel and Ralph E. Johnson. Arithmetic and Double Dispatching in Smalltalk-80. In *Journal of Object-Oriented Programming*, V2 N6 March/April 1990, 40-44

[Ingalls 1978] Daniel H. H. Ingalls. The Smalltalk-76 Programming System: Design and Implementation. In *5th ACM Symposium on POPL,* 1978, 9-15

[Ingalls 1986] D.H.H. Ingalls. A simple technique for handling multiple polymorphism. *In Proceedings of OOPSLA '86,* 1986.

[Johnson 1988b] Ralph E. Johnson, Justin O. Graver, and Laurance W. Zurawski. TS: An Optimizing Compiler for Smalltalk. In *OOPSLA '88 Proceedings*, 1988, 18-26

[Kiczales & Rodriguez 1990] Gregor Kiczales and Luis Rodriguez. Efficient Method Dispatch in PCL. In Proceedings of *the ACM Conference on Lisp and Functional Programming*, 1990, 99-105.

[Kiczales 1991] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow.*The Art of the Metaobject Protocol.* MIT Press, 1991

[Kiczales 2001] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP Proceedngs* 2001.

[Keene 1989] Sonya E. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Introduction to CLOS.* Addison-Wesley, 1989

[Krasner 1983] Glenn Krasner, editor. *Smalltalk 80: Bits of History, Words of Advice.* Addison-Wesley, Reading, MA 1983

[Kristensen 1990] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Language.* 1990

[LaLonde 1986] Wilf R. LaLonde, Dave A. Thomas and John R. Pugh. *An Exemplar Based Smalltalk.* OOPSLA '86 Proceedings . Portland, OR, October 4-8 1977 pages 322-330

[Leavens and Millstein] Multiple Dispatch as Dispatch on Tuples. In *OOPSLA Proceedings*, 1998, 274-287.

[McAffer 1995] Jeff McAffer. Meta-level Programming with CodA. In *ECOOP Proceedings* 1995, 190-214.

[Maes 1987a] Pattie Maes. *Computational Reflection.* Artificial Intelligence Laboratory. Vrije Universiteit Brussel. Technical Report 87-2, 1987

[Maes 1987b] Pattie Maes. Concepts and Experiments in Computational Reflection. *In OOPSLA '87 Proceedings*. 1987, 147-155.

[Moon 1986] David Moon, Object-Oriented Programming with Flavors, In *OOPSLA '86 Proceedings,* 1986 1-8

[Mougin 2003] Philippe Mougin, Stéphane Ducasse: OOPAL: integrating array programming in object-oriented programming. *In OOPSLA Proceedings*, 2003, 65-77

[Mugridge 1991] Warwick Mugridge, John Hamer, John Hosking. Multi-Methods in a StaticallyTyped Programming Language. In *ECOOP Proceedings*, 1991, 147-155

[Paepcke 1993] Andreas Paepcke (editor), Object-Oriented Programming: The CLOS Perspective, MIT Press, 1993

[Palsberg 1998] Jens Palsberg, C. Barry Jay, James Noble. Experiments with Generic Visitors. In *the Proceedings of theWorkshop on Generic Programming,* Marstrand, Sweden, 1998.

[Palsberg 2004] Jens Palsberg and J Van Drunen. Visitor oriented programming. In the Workshop for Foundations of Object-Oriented Programming (FOOL), 2004.

[Rivard 1997] Fred Rivard. *Evolution du comportement des objets dans les langages a classes reflexifs*. PhD thesis, Ecole des Mines de Nantes, France, June 1997

[Saltzer 1981] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design, *Second International Conference on Distributed Computing Systems* (April, 1981) pages 509-512.

[Scharli 2004] Nathanael Schärli, Andrew P. Black, Stéphane Ducasse: Object-oriented encapsulation for dynamically typed languages. In *OOPSLA Proceedings.* 2004, 130-149

[Smith 1983] Brian Cantwell Smith. Reflection and Semantics in Lisp. In *POPL Proceedings*, 1984, 23-35

[Stefik 1986a] Mark Stefik and Daniel G. Bobrow. Object-Oriented Programming: Themes and Variations. *AI Magazine* 6(4): 40-62, 1986

[Stroustrup 1986] Bjarne Stroustrup. *The C++ Programming Language,* Addison-Wesley, Reading, MA, 1986

[Thorup 1997] Thorup, K. K. Genericity in Java with virtual types. In *ECOOP Proceedings* 1997, 444-471.

[Ungar 1987] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *OOPSLA '87 Proceedings.* 1987, 227-242.

Interprocedural Analysis for Privileged Code Placement and Tainted Variable Detection

Marco Pistoia[1], Robert J. Flynn[2], Larry Koved[1], and Vugranam C. Sreedhar[1]

[1] IBM Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA
{pistoia, koved, vugranam}@us.ibm.com
http://www.research.ibm.com/javasec

[2] Polytechnic University, 6 Metrotech Center, Brooklyn, NY 11201, USA
flynn@poly.edu
http://www.poly.edu

Abstract. In Java 2 and Microsoft .NET Common Language Runtime (CLR), trusted code has often been programmed to perform access-restricted operations not explicitly requested by its untrusted clients. Since an untrusted client will be on the call stack when access control is enforced, an access-restricted operation will not succeed unless the client is authorized. To avoid this, a portion of the trusted code can be made "privileged." When access control is enforced, privileged code causes the stack traversal to stop at the trusted code frame, and the untrusted code stack frames will not be checked for authorization. For large programs, manually understanding which portions of code should be made privileged is a difficult task. Developers must understand which authorizations will implicitly be extended to client code and make sure that the values of the variables used by the privileged code are not "tainted" by client code. This paper presents an interprocedural analysis for Java bytecode to automatically identify which portions of trusted code should be made privileged, ensure that there are no tainted variables in privileged code, and detect "unnecessary" and "redundant" privileged code. We implemented the algorithm and present the results of our analyses on a set of large programs. While the analysis techniques are in the context of Java code, the basic concepts are also applicable to non-Java systems with a similar authorization model.

1 Introduction

The Java 2 [28, 29, 17] and Microsoft .NET Common Language Runtime (CLR) [16] programming models are extensively used in different kinds of Internet applications. In such applications, it is essential that, when access to a restricted resource is attempted, all code currently on the call stack is authorized to access that resource. In Java 2, when access to a restricted resource is attempted, the `SecurityManager`, if active, triggers access-control enforcement by invoking `AccessController.checkPermission()`. This method takes a `Permission` object `p` as a parameter and performs a call-stack walk to verify that each caller in the current thread of execution has been granted the authorization represented by `p`. In CLR, the call-stack walk is performed by the `Demand()` method. In both platforms, a `SecurityException` is thrown if the declaring class of any one of the methods on the call stack does not have the appropriate authorization.

Often, however, trusted code has been programmed to perform access-restricted operations—such as writing to a log file—that its untrusted client did not explicitly request. Since the untrusted client will be on the call stack when access control is enforced, the operation will not succeed unless the client code is authorized as well. To avoid authorizing the client, which would constitute a violation of the Principle of Least Privilege [32], the portion of trusted code performing the restricted operation must be made *privileged*. In Java 2, this is done by wrapping that portion of trusted code into a call to `AccessController.doPrivileged()`. In CLR, the same result can be obtained by having the trusted code call the `Assert()` method. When access control is enforced, privileged code causes the call-stack walk to stop at the stack frame where `doPrivileged()` is invoked. As a result, client code is implicitly granted the right to perform the restricted operation while the current thread is executing.

Taking preexisting trusted code and understanding which portions of it should be made privileged is a difficult task. It is even more challenging when the trusted code is large or complex. Besides identifying the blocks of it that require authorizations, developers must understand which access rights the privileged code will implicitly grant to client code, and make sure that the variables used by the privileged code to access restricted resources are not *tainted*, meaning that their values cannot be arbitrarily influenced by the client code [35]. For example, if the privileged code is responsible for logging to a file, the name of the log file should not be tainted. Otherwise, an untrusted caller could invoke that privileged code and modify any file in the file system. As we shall see, a tainted variable can be considered *sanitized* if it satisfies certain preconditions.

This paper presents an interprocedural analysis for Java bytecode to solve the following problems:

1. Identify portions of trusted code that should be made privileged, with three objectives in mind:
   (a) Respect the Principle of Least Privilege by preventing unnecessary authorization requirements from propagating to client code
   (b) Ensure that no unnecessary `SecurityException`s are thrown due to the client's being insufficiently authorized
   (c) Ensure that there are no tainted variables in privileged code, unless they have been previously sanitized
2. Automatically detect if a tainted variable is *malicious* (used inside privileged code to access a restricted resource) or otherwise *benign*
3. Detect existing "unnecessary" and "redundant" privileged blocks of code and avoid introducing new ones

Privileged code is *unnecessary* if there is no path from it to any authorization check, and it is *redundant* if all the authorization checks it leads to are dominated by other privileged code. Unnecessary or redundant privileged code may lead to violations of the Principle of Least Privilege, especially as a result of subsequent code maintenance, and can be expensive from a performance point of view.

The rest of this section further discusses why privileged-code and tainted-variable analysis is important and summarizes the key contributions of this paper.

1.1 Trusted Code Access Control

When client code makes a call into trusted code, the trusted code often accesses restricted resources that the client never intended to, nor does it need to, directly access. For instance, assume that a Java program is authorized to open a network socket. To do so, it invokes `createSocket()` on the `LibraryCode` trusted class in Figure 1. As its code shows, on opening a socket on behalf of a client program, the trusted code is programmed to log the socket operation to a file. According to the Java 2 access-control model, both the trusted code and its client will need to be granted the `FilePermission` to modify the log file and the `SocketPermission` to create the socket connection, even though the client did not explicitly request to write to the log file. Granting the client code the access right to modify the log file would violate the Principle of Least Privilege. One way to circumvent this problem is to mark the portion of the trusted code responsible for logging as privileged. This prevents the call-stack inspection for the log operation from going beyond the `createSocket()` method, and temporarily exempts the client from the `FilePermission` requirement during the execution of `createSocket()`.

From a practical point of view, a Java developer must implement either the `PrivilegedExceptionAction` or `PrivilegedAction` interface, depending on whether the privileged code could throw a checked `Exception` or not, respectively. Both these interfaces have a `run()` method that, once implemented,

```
import java.io.*;
import java.net.*;
public class LibraryCode {
    private static String logFileName = "audit.txt";
    public static Socket createSocket(String host, int port)
            throws UnknownHostException, IOException {
        // Create the Socket
        Socket socket = new Socket(host, port);
        // Log the Socket operation to a file
        FileOutputStream fos = new FileOutputStream(logFileName);
        BufferedOutputStream bos = new BufferedOutputStream(fos);
        PrintStream ps = new PrintStream(bos, true);
        ps.print("Socket " + host + ":" + port);
        return socket;
    }
}
```

Fig. 1. Library Code Propagating Authorization Requirements to Its Clients

```
import java.io.*;
import java.net.*;
import java.security.*;
public class LibraryCode2 {
    private static final String logFileName = "audit.txt";
    public static Socket createSocket(String host, int port) throws
            UnknownHostException, IOException, PrivilegedActionException {
        // Create the Socket
        Socket socket = new Socket(host, port);
        // Log the Socket operation to a file using doPrivileged()
        File f = new File(logFileName);
        PrivWriteOp op = new PrivWriteOp(host, port, f);
        FileOutputStream fos = (FileOutputStream)
                AccessController.doPrivileged(op);
        BufferedOutputStream bos = new BufferedOutputStream(fos);
        PrintStream ps = new PrintStream(bos, true);
        ps.print("Socket " + host + ":" + port);
        return socket;
    }
}
class PrivWriteOp implements PrivilegedExceptionAction {
    private File f;
    PrivWriteOp (File f) {
        this.f = f;
    }
    public Object run() throws IOException {
        return new FileOutputStream(f);
    }
}
```

Fig. 2. Library Using Privileged Code

must contain the portion of trusted code performing the restricted operation not directly requested by the client. Next, the `PrivilegedExceptionAction` or `PrivilegedAction` instance is passed as a parameter to the `doPrivileged()` method, which will invoke the instance's `run()` method. Class `LibraryCode2` in Figure 2 is obtained by modifying class `LibraryCode` in Figure 1. The main modification consists of wrapping the call to the `FileOutputStream` constructor in a privileged block to prevent client code from requiring a `FilePermission`.

Frequently, code is not written with security as a concern, or it is written to run on a version of the Java Runtime Environment (JRE) prior to 1.2.[1] When a Java 2 `SecurityManager` is finally turned on for a particular application, `SecurityExceptions` are thrown due to access control violations. It can be very difficult to understand which portions of trusted code should be made privileged. In practice, this problem is solved empirically. The developer tests the trusted code with sample client code that makes calls into the trusted code. Typically,

[1] The Java 2 fine-grained access control model was introduced in version 1.2.

the client code is granted only a limited number of access rights, while the trusted code is granted sufficient authorizations, such as `AllPermission`. The developer then notes all the `SecurityException`s generated when running the test cases and distinguishes between two categories of `SecurityException`s:

1. The `SecurityException`s due to the client code's attempting to access some protected resources through the trusted code without the adequate authorizations
2. The `SecurityException`s due to the trusted code's attempting to access some restricted resources on its own without using privileged code

Eliminating a `SecurityException` of Category 2 requires inspecting the trusted source code, identifying which portion of it is responsible for accessing the restricted resource, and making that portion privileged. A `SecurityException` of Category 1 can instead be eliminated by granting the client code the necessary access rights, but this operation must be performed cautiously because granting authorizations to the client could hide `SecurityException`s of Category 2. Manually performing this task is difficult, tedious, and error-prone. After modifying the trusted code or the client security policy, the developer must rerun the test cases. This process must be repeated, possibly many times, until there are no more authorization failures. Additionally, `doPrivileged()` requirements in the trusted code may remain undiscovered due to an insufficient number of test cases, which makes production code potentially unstable.

1.2 Tainted Variables

Another security concern when inserting `doPrivileged()` calls is the risk that the privileged code uses tainted variables to access restricted resources. Consider, for example, the `GetSocket()` utility class shown in Figure 3. Both `host` and `port` are tainted variables, since an untrusted client can arbitrarily set them. Their use in privileged code to open a socket makes them a potential security risk. Conversely, variable `userName`, though tainted and used in privileged code, is benign since its value is not used to access a restricted resource.

Sometimes, it may be necessary to use tainted variables inside privileged code to access restricted resources. In such cases it is important to perform *sanity checks* on those variables to verify that they satisfy certain preconditions [3]. For example, in the code of Figure 3, the programmer could *sanitize* `host` and `port` and make them untainted by refusing to execute the privileged code if, for example, the `host` value does not end with `.edu` and the `port` value is different from `443`.

In general, manually ensuring that no malicious tainted variables are used inside privileged code is time consuming and error prone. It requires:

1. Identifying all the unsanitized malicious tainted variables (`host` and `port` in Figure 3) and separate them from the benign ones (`userName`)
2. Determining all the control- and data-flow paths in the execution of the program that would allow an unsanitized malicious tainted variable to be used inside some privileged code to access restricted resources

```
import java.net.*;
import java.security.*;
public class GetSocket {
    public static Socket getSocket(final String host, final int port,
            final String userName) throws Exception {
        Socket s;
        PrivOp op = new PrivOp(host, port, userName);
        try {
            s = (Socket) AccessController.doPrivileged(op);
        }
        catch (PrivilegedActionException e) {
            throw e.getException();
        }
        return s;
    }
}
class PrivOp implements PrivilegedExceptionAction {
    private String host, userName;
    int port;
    PrivOp(String host, int port, String userName) {
        this.host = host;
        this.port = port;
        this.userName = userName;
    }
    public Object run() throws Exception {
        System.out.println("Received request from user " + userName);
        return new Socket(host, port);
    }
}
```

Fig. 3. Helper `getSocket()` Method with Tainted Parameters

Having a tool that automatically determines if code candidate to become privileged uses unsanitized, malicious tainted variables to access restricted resources helps when deciding whether making that code privileged is appropriate. For example, the code of Figure 1 has two instructions that could be made privileged:

1. `Socket socket = new Socket(host, port);`
2. `FileOutputStream fos = new FileOutputStream(logFileName);`

If Instruction 1 is made privileged, then parameters `host` and `port`, which are tainted, will constitute a security exposure since they are used to access a restricted resource. This is an indication that Instruction 1 should not be made privileged. Conversely, parameter `logFileName` is not tainted. This is an indication that Instruction 2 could be made privileged.

1.3 Contributions

From a privileged-code analysis perspective, the set of code components involved in the execution of a program is logically partitioned into three disjoint subsets:

1. The *fixed components*, which normally include the JRE libraries and are not suitable, or candidates, for modification
2. The *modifiable components*, which are those considered for modification and privileged-code placement, and are typically trusted
3. The *client components*, which make calls into the modifiable components and are often not available at analysis time

This paper presents an interprocedural privileged-code placement and tainted-variable analysis algorithm. The algorithm assumes that both the sets of fixed and modifiable components are available to the analysis, whereas the presence of the client components is statically modelled. The interprocedural analysis described in this paper achieves the following results:

1. For each authorization check triggered by a modifiable component, the analysis identifies the modifiable component's code location that, from a control-flow perspective, is the closest to the authorization check, which minimizes the risks of violating the Principle of Least Privilege.
2. Code locations candidate for becoming privileged are identified with a precision that goes to the level of the program counter within a method (and the source-code line number where available).
3. The analysis provides an explanation as to why a call to `doPrivileged()` is recommended or not.
4. The analysis detects which authorizations will be implicitly granted to client code as a result of calling `doPrivileged()`.
5. The analysis minimizes the risks of introducing unnecessary or redundant privileged code in a modifiable component.
6. If unnecessary or redundant privileged code is already present in a modifiable component, the analysis will detect it.
7. The analysis distinguishes between malicious and benign tainted variables.
8. The analysis detects if unsanitized malicious tainted variables are used in privileged code to access restricted resources.

We implemented this analysis framework in a security-analysis tool called Mandatory Access Rights Certification of Objects (MARCO). In this paper, we present our experience in using MARCO on a set of components, some of which contained more than 20,000 classes. While the analysis techniques described in this paper are in the context of Java code, the basic concepts are also applicable to privileged-code placement issues in non-Java systems, including CLR.

1.4 Organization of This Paper

Section 2 introduces the control- and data-flow frameworks on which the MARCO tool is based. Section 3 describes an *access-rights analysis algorithm* for computing authorization requirements of Java code. Section 4 shows how the access-rights analysis algorithm can be enhanced to compute modifiable-component code locations that are closest to the authorization checks. The *privileged-code placement algorithm* described in Section 4 minimizes the chances of introducing unnecessary or redundant privileged code. Additionally, if unnecessary or redundant privileged code is already present, the algorithm will detect it. Section 5 presents *a tainted-variable analysis algorithm* for detecting potential misuses of tainted variables in code that is already privileged, or that is a candidate for becoming privileged as a result of executing the privileged-code placement algorithm. Section 6 presents our experience with running the MARCO tool on complex commercial-quality code. Section 7 describes previous results in the area of authorization, privileged-code, and tainted-variable analysis, and explains why the work presented in this paper is innovative with respect to those results. Finally, Section 8 summarizes the most important results presented in this paper.

2 Foundations of the Analysis Framework

The first analysis step is to construct an augmented, domain-specific invocation graph called an Access-Rights Invocation Graph (ARIG) [24]. An ARIG is a directed multi-graph $G = (N, E)$, where N is a set of nodes and E is a set of edges, with the following characteristics:

- Each node in the graph:
  - Represents a context-sensitive method invocation.
  - Is uniquely identified by its *calling context*:
    - The target method
    - The receiver and parameters values
  - Contains the following *state*:
    - The target method
    - For instance methods, an allocation site for the method's receiver
    - All parameters to the method, represented as a vector of sets of possible allocation sites
    - A set of possible return values from the target method, represented as a set of allocation sites
  - Is associated with a class loader name, corresponding to the name of the class loader that would load the method's declaring class at run time.
- Each labelled[2] edge $e = (m, n) \in E$ points from a call site in the method represented by node m to the target method represented by node n.
- An ARIG allows for bidirectional traversal.

[2] For simplicity, in this paper, we indicate the edges of an ARIG $G = (N, E)$ as pairs of nodes of the form (m, n), where $m, n \in N$. However, the edges of G are actually triplets of the form (m, n, w), where $m, n \in N$ and w is a call site in the method represented by m and pointing to the target method represented by n. In this sense, G is a multi-graph because there may exist multiple edges between any two nodes m and n, and those edges are distinguishable from each other based on the call-site information, which acts as a *label*. The call-site information contains the program counter at which w occurs.

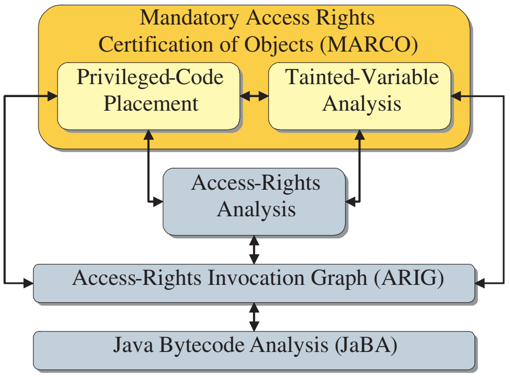

Mandatory Access Rights
Certification of Objects (MARCO)
Privileged-Code Placement
Tainted-Variable Analysis
Access-Rights Analysis
Access-Rights Invocation Graph (ARIG)
Java Bytecode Analysis (JaBA)

Fig. 4. Architecture of the Analysis Framework

An ARIG is used to execute an access-rights analysis. The results of the access-rights analysis and the ARIG itself are used by the MARCO tool to perform privileged-code placement and tainted-variable analysis. As Figure 4 shows, an ARIG is constructed using the Java Bytecode Analysis (JaBA) framework, which adopts a Control-Flow Analysis (CFA) [25] disambiguating between heap objects according to their allocation sites, with extra context for `Permission` objects. Specifically, JaBA is:

– *Path insensitive* [19] because it does not evaluate conditional statements and conservatively assumes that each conditional branch out of a conditional statement will be executed
– *Intraprocedurally flow sensitive* [31] because it considers the order of execution of the instructions within each basic block, accounting for local-variable kills [22] and casting of object references
– *Interprocedurally flow insensitive* [31] because it uses the conservative assumption that all instance and static fields are subject to modification at any time due to multi-threading
– *Context sensitive* [31] because its interprocedural analysis uniquely distinguishes each node by its calling context, with a context-sensitivity policy similar to Agesen's Cartesian Product Algorithm (CPA) [1]
– *Field sensitive* [31] because an object's fields are represented distinctly

An ARIG is domain-specific in that it is tailored to access-rights analysis, privileged-code placement, and tainted-variable analysis needs. Its domain-specific characteristics are described in the remainder of this section.

2.1 Modelling Multi-threading

In Java 2, when access to a restricted resource is attempted from within a child thread, all the code in the child thread and in all its ancestor threads must be granted the right to access that resource. This behavior can be modelled by identifying all the `run()` nodes in G whose receiver is a `Thread` object. For each of such nodes r, with receiver `t`, the node c representing the invocation of the `Thread` constructor that instantiated `t` in the parent thread is identified, and a new edge (c, r) is added to E. At the same time, the edge (s, r), where s represents the invocation of `start()` on `t`, is removed from E.

2.2 Extra Context for `Permission` Objects

The `Permission` parameter passed to `AccessController.checkPermission()` is frequently instantiated by the `SecurityManager`. For example, when the `SecurityManager`'s `checkWrite()` method is invoked, it instantiates a `FilePermission` and passes it to the `SecurityManager`'s `checkPermission()` method, which finally passes it to `AccessController.checkPermission()`. One problem is that different `FilePermission` objects instantiated through calls to `checkWrite()` in different parts of the program will all share the same type and allocation site. Therefore, JaBA would represent them as if they were the same object, yielding overly conservative results. The solution we adopted was to add extra context to `Permission` objects. Specifically, the context used to represent a `Permission` object `p` is not just the type and the allocation site of `p`, but also the node containing the allocation site of `p`. Therefore, if $m, n \in N$ are `checkWrite()` nodes in the ARIG such that the parameters for the method calls they represent are the `String`s `file1` and `file2`, respectively, the `FilePermission` allocated in m will be distinguished from the one allocated in n because $m \neq n$, even though both `FilePermission`s share the same type and allocation site. To avoid building an unnecessarily large invocation graph, this specialization is only applied to `Permission` objects allocated in the `SecurityManager`.

2.3 Propagation of String Constants

The constructor of a `Permission` object `p` takes zero or more `String` objects as parameters. As we shall see, the fully qualified `Permission` class name and the `String`s passed to the constructor of `p` uniquely identify the authorization requirement represented by `p`. For Java 2 authorization-related analyses, keeping track of string constants is, therefore, essential. For this reason, an ARIG includes propagation of string constants, unless these are dynamically generated.

2.4 Modelling of Callbacks

When building an invocation graph modelling the execution of a program, the method entry points can have parameters, which may include the receiver object, `this`. JaBA offers two options:

1. If the modifiable components being analyzed are part of a self-contained application, the analysis is typically treated as a *closed-world analysis*—one in which all the code executed at run time is also available during the analysis. In this case, JaBA uses Class Hierarchy Analysis (CHA) [11] to build the class hierarchy rooted at the parameter's declared type. When a callback from

that parameter object is encountered, JaBA models it by looking for all the possible implementations of the invoked method in the class hierarchy.

2. If the modifiable components under analysis are part of a library, the analysis is said to be an *open-world* or *incomplete-program analysis* [31] because the values and object sources of those parameters are part of the client application, which typically is only available at run time, unless the declared types of those parameters are final. If a callback from a parameter object of a non-final type occurs, JaBA, conservatively, does not model it because no control- and data-flow details on that callback are available at analysis time. However, JaBA records that a callback has been encountered. Potentially, each of such callbacks could require `AllPermission` at run time.

3 Access-Rights Analysis for Privileged Code

In this section, we present a simple data-flow analysis model for propagating access-rights and privileged-code requirements along an ARIG $G = (N, E)$. To compute the portions of modifiable-component code that should be made privileged, it is necessary to statically model the Java 2 authorization subsystem. Recall that, in Java 2, the run time enforces authorization by ultimately making a call to `checkPermission()` with a parameter `p` of type `Permission` representing the resource access being attempted. From what we said in Section 2.3, for authorization purposes, `p` can be characterized solely based on `p`'s *permission ID*, which consists of `p`'s fully-qualified class name and the `String` instances used to instantiate `p`.[3] For example, if `p` was instantiated with the statement

```
Permission p = new java.io.FilePermission("audit.txt", "write")
```

then `p`'s permission ID is `java.io.FilePermission "audit.txt", "write"`.

Let P be the universe of all the permission IDs associated with the code being analyzed. A function $\Pi : N \to 2^P$ can be defined that maps each node $n \in N$ to the set of permission IDs representing the `Permission` objects necessary to execute the method represented by n. Permission IDs represent authorization requirements. Determining privileged-code placement in a set of modifiable components involves propagating permission IDs across the ARIG representing the execution of those components.

3.1 Identification of `checkPermission()` Nodes

The first step of the algorithm is to iterate over all the nodes of G to identify those that correspond to `checkPermission()` method calls. For each of such nodes a, all the possible `Permission` allocation sites that have flowed to the

[3] In fact, in the JRE reference implementation, authorizations are granted to programs and principals by just listing the corresponding permission IDs in a flat-file policy database, called the *policy file* [29].

formal argument are identified,[4] and the permission IDs are computed from the corresponding constructor nodes. This phase requires $\mathcal{O}(|N|)$ time.

3.2 Reverse Propagation of Permission IDs

The Java 2 authorization subsystem mandates that, at the point where `checkPermission()` is invoked with a `Permission` parameter `p`, all the code on the execution thread's stack be granted the authorization represented by `p`. This can be modelled by identifying the node a corresponding to the `checkPermission()` call and propagating the permission ID corresponding to `p` backwards to all the predecessors of a, recursively. Thus, each node $n \in N$ is mapped to a (possibly empty) set of permission IDs, obtained as the union of the permission ID sets propagated from n's successors as follows:

$$\Pi(n) = \bigcup_{m \in Succ(n)} \Pi(m)$$

where $Succ(n) = \{m \in N | (n, m) \in E\}$. When for some $n \in N$, $\Pi(n)$ changes as a result of this propagation, $\Pi(m)$ is unioned with $\Pi(n)$ for all $m \in Pred(n)$, where $Pred(n) = \{m \in N | (m, n) \in E\}$.

The reverse propagation of permission ID sets just described can be formalized in terms of data flow. Using a standard data-flow notation [22, 2, 25], we define *data-flow sets* $GEN(n)$ and $KILL(n)$ for each node $n \in N$ as follows:

- $GEN(n)$ contains the permission IDs *generated* by node n. Such permission IDs correspond to the authorizations checked at the method represented by node n. For the Java 2 access-control model, $GEN(a) \neq \varnothing$ if and only if a is a `checkPermission()` node. In particular, for any such node a, $GEN(a)$ contains exactly the permission IDs corresponding to the authorizations checked at the method represented by a in the ARIG.
- $KILL(n)$ contains the permission IDs *killed* by node n. Such permission IDs correspond to authorization requirements whose propagations on the call stack stop at the predecessors of node n. According to the Java 2 access-control model, if $d \in N$ represents a `doPrivileged()` method invocation, $KILL(d)$ is the universe P of all the permission IDs defined in the ARIG. This is because, in Java 2, a call to `doPrivileged()` does not extend authorizations to client code selectively, in a fine-grained fashion, but does it in a coarse-grained fashion.[5] For any other node $n \in N$, $KILL(n) = \varnothing$.

[4] Even though `checkPermission()` takes only one `Permission` parameter, that parameter may correspond to more than one object in the ARIG model, since JaBA is path insensitive and interprocedurally flow insensitive.

[5] Unlike the Java 2 `doPrivileged()` method, the CLR `Assert()` method shields client code from authorization requirements in a fine-grained fashion [7]. Library code can assert a specific `IPermission` object, and only the authorization represented by that object will be implicitly granted to the client code currently on the stack. To model this behavior correctly, the $KILL$ set of the asserting method's node would only have to contain the permission IDs of the asserted `IPermission` objects.

It is therefore convenient to introduce a function $NodeType : N \rightarrow \{check, grant, other\}$. For each $n \in N$, $NodeType(n)$ is defined as follows:

$$NodeType(n) = \begin{cases} check, \text{ if } n \text{ is a } \texttt{checkPermission()} \text{ node;} \\ grant, \text{ if } n \text{ is a } \texttt{doPrivileged()} \text{ node;} \\ other, \text{ otherwise.} \end{cases}$$

The *check* nodes are those representing `checkPermission()` method calls, which trigger authorization checks, while the *grant* nodes are those representing calls to `doPrivileged()`, through which the callers on the thread stack are implicitly granted authorizations. The *other* nodes do not affect the data flow. The following pseudo-code formalizes the assignment of the data-flow sets:

```
1:  for each node $n$ {
2:    switch($NodeType(n)$) {
3:      case $check$ :
4:        $GEN(n) = \{p \in P | p \text{ is checked at } n\}$
5:        $KILL(n) = \varnothing$
6:      case $grant$ :
7:        $GEN(n) = \varnothing$
8:        $KILL(n) = P$
9:      case $other$ :
10:       $GEN(n) = \varnothing$
11:       $KILL(n) = \varnothing$
12:   }
13: }
```

The data-flow equations for each node $n \in N$ are defined in the usual way as follows:

$$\begin{aligned} OUT(n) &= (IN(n) \cup GEN(n)) - KILL(n) \\ IN(n) &= \bigcup_{m \in Succ(n)} OUT(n) \end{aligned}$$

where $OUT(n)$ and $IN(n)$ are the sets of permission IDs propagated from n and reaching n, respectively. The data-flow analysis just described converges to a fixed point in $\mathcal{O}(|E||P|)$ time since $(2^P, \subseteq)$ is a finite lattice and the *data-flow functions* $OUT, IN : N \rightarrow 2^P$ are monotonic with respect to the lattice's partial order, $\subseteq$ [18].

3.3 Permission ID Propagation from `doPrivileged()` Nodes

According to the Java 2 authorization subsystem, authorization requirements propagated upwards via a `doPriviledged()` node must not propagate beyond the predecessors of the `doPrivileged()` node. This can be modelled as follows: When a `doPrivileged()` node d is encountered during the reverse propagation of permission IDs described in Section 3.2, its permission ID set, $\Pi(d)$, is propagated to d's predecessors only after the propagation algorithm for all the other nodes has terminated. The propagation of $\Pi(d)$ upwards must not be performed recursively. If n is a node in $Pred(d)$ and $\Pi(n)$ changes as a result of the propagation of $\Pi(d)$ from d, $\Pi(n)$ is not transmitted to the nodes in $Pred(n)$. One data-flow equation is sufficient to describe this one-step propagation:

$$IN(n) = IN(n) \cup \bigcup_{\substack{d \in Succ(n) \\ NodeType(d)=grant}} IN(d)$$

This equation has an effect only for those nodes that have a *grant* node as a successor. The time complexity of this one-step propagation is $\mathcal{O}(|E|)$.

3.4 Complexity

The access-rights analysis converges in $\mathcal{O}(|E||P|)$ time. When the analysis terminates, for each node $n \in N$, the data-flow set $IN(n)$ will be equal to the set $\Pi(n)$ and will represent the authorizations required to execute the method represented by n with n's calling context.

4 Privileged-Code Placement

This section describes how the propagation algorithm described in Section 3 can be augmented to automatically detect which portions of modifiable-component code should be made privileged while minimizing the risks of violating the Principle of Least Privilege, with a precision that goes to the level of the program counter within a method. For each privileged-code location it recommends, the algorithm provides an explanation. Additionally, this section shows how to compute the authorizations that privileged code will implicitly grant to client code. Finally, this section describes how the algorithm detects existing unnecessary or redundant privileged code, and avoids inserting new privileged code that is unnecessary or redundant.

4.1 Insertion of `doPrivileged()` Calls

In Java 2, class loaders are organized as a tree T, called the *class-loading delegation tree* [28]. JaBA models the class-loading system and associates a class loader name with every node in an ARIG. The privileged-code placement process is configured by assuming that all the classes in the modifiable components will be loaded by a designated class loader, called the *component loader*. A *boundary edge* in G is any edge $e = (m, n) \in E$ such that m is associated with the component loader and n is associated with a different class loader in T. If $\Pi(n) \neq \varnothing$, then the call represented by e is guaranteed to lead to the Java 2 authorization subsystem. Such a call is a candidate for becoming privileged. For example, e may be the edge resulting from calling the constructor of `FileOutputStream`

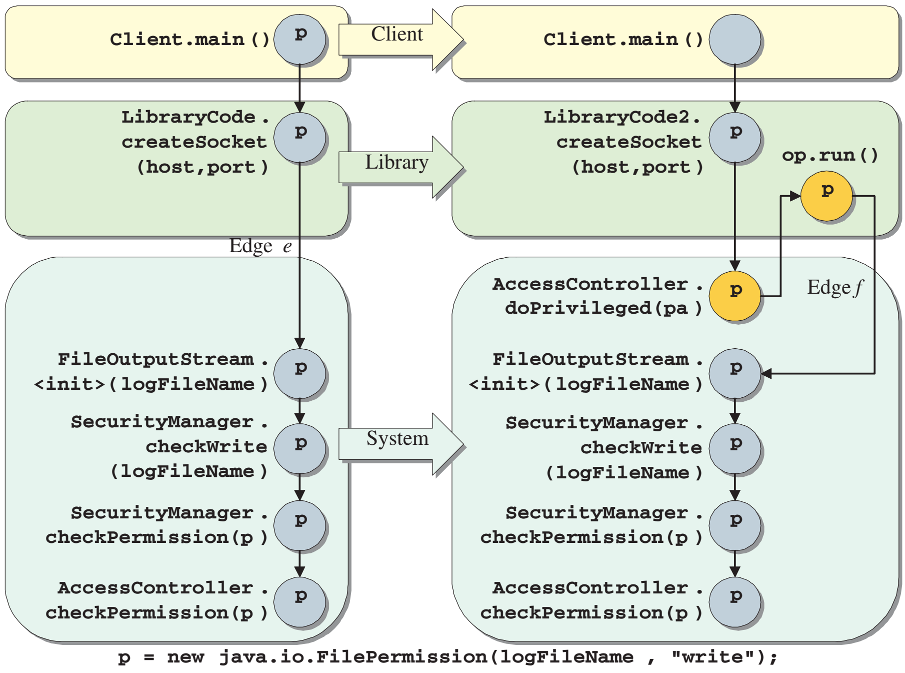

Client.main()
Client
Client.main()
LibraryCode.createSocket(host,port)
Library
LibraryCode2.createSocket(host,port)
op.run()
Edge e
Edge f
AccessController.doPrivileged(pa)
FileOutputStream.<init>(logFileName)
SecurityManager.checkWrite(logFileName)
System
SecurityManager.checkPermission(p)
AccessController.checkPermission(p)
p
p = new java.io.FilePermission(logFileName , "write");

Fig. 5. Changes in the ARIG after Making Library Code Privileged

from method `createSocket()` in Figure 1. Figure 2 shows how to wrap the `FileOutputStream` constructor call into a privileged block, and Figure 5 shows the corresponding ARIGs. Notice how the ARIG on the right in Figure 5 reflects the presence of `doPrivileged()` by not propagating the `FilePermission` requirement beyond the invocation of `createSocket()`, exempting client code from the `FilePermission` requirement, as desired.

The algorithm described in Section 3 can be augmented to identify any boundary edge $e = (m, n)$ such that $\Pi(n) \neq \varnothing$. The information contained in e and $\Pi(n)$ is sufficient to determine the exact portion of modifiable-component code that is a candidate for becoming privileged along with an explanation, and to identify the authorizations that the privileged code will implicitly grant to client code:

1. The class name, method signature, and program counter that constitute a possible `doPrivileged()` location can be obtained from node m and the call site in e. Since e is a boundary edge, from a control-flow perspective the location computed by this algorithm is, in the modifiable-component code, the closest to the authorization check. This ensures that only the portion of modifiable-component code effectively leading to an authorization check will be made privileged, which minimizes the risks of violating the Principle of Least Privilege.

2. The authorizations implicitly granted to clients if a call to `doPrivileged()` is inserted are represented by the permission IDs in $\Pi(n)$.
3. As an explanation, the fully qualified signature of the method being invoked at node n and causing the authorization requirements in $\Pi(n)$ can be obtained from node n itself. If a more detailed explanation is desired, all the paths from n to the `checkPermission()` nodes in the ARIG subgraph rooted at n can be reported. Such paths are those through which the authorization requirements in $\Pi(n)$ have propagated up to n.

The privileged-code placement algorithm can be customized. Instead of recommending the privileged-code locations that, in the modifiable components, are the closest to the authorization checks, the algorithm could, for example, identify the privileged-code locations closest to those components' entry points. With this approach, however, code not requiring authorizations may become unnecessarily privileged.

4.2 Detecting Unnecessary or Redundant Privileged Code

An unnecessary `doPrivileged()` call may result from changes made to modifiable-component code during code development or maintenance. A call to `doPrivileged()` that was originally considered necessary no longer triggers an authorization check after the change. A redundant `doPrivileged()` call may result from poor code design or from integrating different components so that a call to `doPrivileged()` that was once considered necessary because it led to an authorization check becomes redundant because other `doPrivileged()` calls now dominate the authorization check. As we observed in Section 1, unnecessary or redundant privileged code should be made unprivileged for security and performance reasons. The algorithm described in Section 3 can be augmented to identify unnecessary or redundant calls to `doPrivileged()` by simply detecting any `doPrivileged()` node d in the graph such that $\Pi(d) = \varnothing$.

If a code instruction does not require authorizations, it is a poor security practice to make it privileged [35]. The following instruction in the `run()` method of Figure 3 has been made privileged even though it does not access a restricted resource:

```
System.out.println("Received request from user " + userName);
```

Such an instruction should be made unprivileged even though, in this case, $\Pi(d) \neq \varnothing$. The privileged-code placement algorithm can easily detect unnecessarily privileged instructions. Let d be a `doPrivileged()` node and r its `PrivilegedAction` or `PrivilegedExceptionAction` `run()` successor. If $\Pi(r) \neq \varnothing$ and there exists $n \in Succ(r)$ such that $\Pi(n) = \varnothing$, then the method invocation represented by n should be made unprivileged.

4.3 Avoiding Unnecessary or Redundant Privileged Code

The permission ID set $\Pi(n)$ associated with the head node n of a boundary edge $e = (m, n)$ must be non-empty for the privileged-code placement algorithm to recommend a call to `doPrivileged()`. Therefore, except for those cases in which the access-rights analysis conservatively reports unrealizable authorization requirements, none of the `doPrivileged()` calls recommended by the privileged-code placement algorithm are unnecessary or redundant. Furthermore, since the privileged-code placement algorithm precisely identifies the method invocations that should be made privileged, no code instruction will become unnecessarily or redundantly privileged as a result of executing the algorithm, except, again, for the cases of conservativeness.

However, it should be observed that, if the analysis is performed after a call to `doPrivileged()` has been inserted, any edge from the `PrivilegedAction` or `PrivilegedExceptionAction`'s `run()` node into a different class loader's name space will also be, by definition, a boundary edge. For example, in Figure 5, after the `FileOutputStream` constructor has been wrapped in a privileged block, the edge f from the `op.run()` node to the `FileOutputStream.<init>()` node is a boundary edge, and the permission ID set associated with the head node is non-empty. Reporting a privileged-code requirement would make the existing call to `doPrivileged()` redundant. To avoid this situation, any boundary edge originating from a `PrivilegedAction` or `PrivilegedExceptionAction`'s `run()` node such that the only predecessors of the `run()` node are `doPrivileged()` nodes is automatically excluded a candidate for `doPrivileged()`.

4.4 Complexity

The privileged-code placement algorithm is obtained by augmenting the access-rights analysis algorithm in a way that does not affect the algorithm's complexity and convergence except for a constant factor. Therefore, executing the privileged-code placement algorithm still requires $\mathcal{O}(|E||P|)$ time.

5 Tainted-Variable Analysis

We refer to the data that either originate from an untrusted source or that can be derived from an untrusted source as being *tainted* [27]. Tainted data and the variables that hold or reference it can be used for certain kinds of overwrite attacks [27], such as overwriting the name of a file or jump address. Sometimes, however, it may be necessary to use a tainted variable when accessing restricted resources. In such cases, the data can be *sanitized* and made untainted by performing sanity checks on it before using it in restricted operations [3]. Sanity checks are usually domain or component specific. We assume that, for a specific application, there is an associated library containing the sanity checks on that application's tainted variables.

A tainted variable is not necessarily a security problem. It may constitute a security problem if it is also a *privileged* variable, meaning that it is used inside privileged code [35]. Even a privileged tainted variable is not necessarily a security problem. In fact, we distinguish two types of privileged tainted variables: if a privileged tainted variable is used to access a restricted resource, we will call it *malicious*, otherwise we will call it *benign.* Since authorization checks are not performed beyond the stack frame invoking `doPrivileged()`, an untrusted client application could exploit a malicious variable to have the privileged code access restricted resources on its behalf.

In Figure 2, variable `logFileName` is not tainted because its value cannot be set by a client application. In Figure 3, the `host` and `port` parameters are both tainted because their values can be set by any client application and no sanity check is performed on them. Additionally, they are both privileged because they are used inside privileged code, and they are malicious because they are are used to access a restricted resource. An untrusted client, with no `SocketPermission`, can invoke `getSocket()` on the trusted library and have the library open an arbitrary socket connection on its behalf. Variable `userName`, though tainted and privileged, is benign because it is not used to perform any restricted operation.

This section presents a simple interprocedural tainted-variable analysis algorithm that augments the privileged-code placement algorithm described in Section 4. The objective of the tainted-variable analysis is both to detect existing malicious variables and to avoid the introduction of new malicious variables when making new code privileged.

The first step of the tainted-variable analysis algorithm is to compute the initial set S of tainted variables, which is the union of the following two sets:

- Set S_1, containing the modifiable-component instance and static fields that can be modified by client code
- Set S_2, containing all the parameters to the modifiable components' public and protected entry methods, including the receiver objects for non-static methods

Set S_2 can be computed easily. However, it should be observed that if a package in a modifiable component is not sealed [29], then S_2 should contain the parameters to the package's default-scope methods as well.

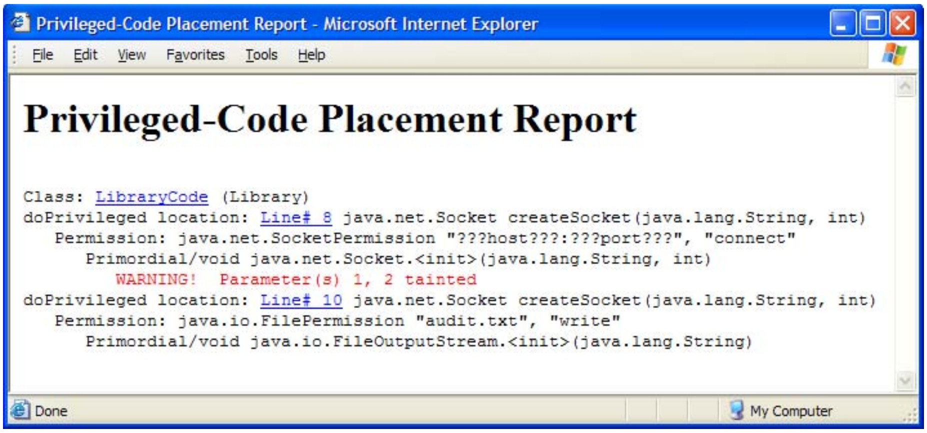
A tainted variable can, directly or indirectly, taint other variables, for example through assignments. The second step of the tainted-variable analysis algorithm consists of identifying existing privileged variables in the modifiable components and detect if they are tainted. Any standard interprocedural program-slicing analysis algorithm [36] can be used to detect value flows from tainted variables to privileged variables. Our algorithm uses a program-slicing algorithm that, for any privileged variable `x`, constructs a slice of `x` and then checks if the slice contains variables in set S. If so, `x` is potentially tainted as well. It remains to be seen whether `x` is benign or malicious. The access-rights analysis algorithm gives us the answer. Since `x` is privileged, `x` must be used in at least one privileged instruction. For `x` to be benign, it must be $\Pi(n) = \varnothing$ for any node n representing a method executing any privileged instruction that contains `x`. If there exists a node n representing a method in which a privileged instruction containing `x` is executed, such that $\Pi(n) \neq \varnothing$, then we conservatively assume that `x` is malicious, in which case we look for a sanity check on `x`. If a sanity check for `x` exists and it can be determined that `x` passes it, then `x` is considered sanitized. Otherwise, the tainted-variable analysis reports a potential security risk.

If `x` is not an existing privileged variable, but it would become so by making new code privileged as a result of executing the privileged-code placement algorithm, the tainted-variable analysis proceeds exactly as before. The only difference is that it will report that the code containing x can be made privileged only if `x` is benign or sanitizable.

6 Experimental Results

Context- and intraprocedurally flow-sensitive static analysis has a reputation for requiring significant processing power and memory. We have performed privileged-code and tainted-variable analysis on parts of `rt.jar`, large commercial middleware, and the Standard Performance Evaluation Corporation Java Business Benchmark 2000 (SPECjbb2000) program [34]. The simplest library that we analyzed is the `LibraryCode` class in Figure 1. Figure 6 shows the HyperText Markup Language (HTML) output produced by the MARCO tool. For better usability, the HTML output has links to the source code anchored to the line numbers where a call to `doPrivileged()` should be inserted. As expected, the tool reported two possible privileged-code placements, but detected that both the parameters to the `Socket` constructor were tainted. This was an indication that only the call to the `FileOutputStream` constructor could be safely made privileged. The results reported by the MARCO tool on all the other benchmarks were also correct based on source-code manual inspection and subsequent testing. In particular, the tool detected when existing privileged code was unnecessary or redundant, and appropriately distinguished between malicious and benign tainted variables.

Most recently, we analyzed Eclipse V3.0 [13] to identify which portions of the plug-in code should be made privileged in order to enable Eclipse to run


Privileged-Code Placement Report - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Done
My Computer

Privileged-Code Placement Report

```
Class: LibraryCode (Library)
doPrivileged location: Line# 8 java.net.Socket createSocket(java.lang.String, int)
   Permission: java.net.SocketPermission "???host???:???port???", "connect"
      Primordial/void java.net.Socket.<init>(java.lang.String, int)
         WARNING!  Parameter(s) 1, 2 tainted
doPrivileged location: Line# 10 java.net.Socket createSocket(java.lang.String, int)
   Permission: java.io.FilePermission "audit.txt", "write"
      Primordial/void java.io.FileOutputStream.<init>(java.lang.String)
```

Fig. 6. Privileged-Code Placement Report on `LibraryCode`

Table 1. Analysis Results

Eclipse Plug-in	*Classes*	*Methods*	*Time (sec)*	*Nodes*	*Edges*	*Instr. (bytes)*	*doPriv.*
`ant`	2245	14799	4668	169539	1833305	818342	1908
`core.runtime`	1265	7233	1379	59771	134191	421094	353
`osgi`	1069	6091	814	62031	141397	362482	256
`tomcat`	2804	19885	5793	197709	471957	1066369	2011
`ui`	2910	16299	11254	191752	1843518	891270	150
`ui.forms`	972	4497	4430	29199	59605	286739	14

with a Java 2 `SecurityManager` enabled. The results reported in Table 1 are from running the MARCO tool on an IBM Personal Computer with an Intel 1.6 GHz Pentium M processor and 1 GB of Random Access Memory (RAM), and with operating system Microsoft Windows XP SP2. The MARCO tool has been implemented in Java. We ran it both as a stand-alone application and inside Eclipse V3.0 using Sun Microsystems' JRE V1.4.2_02. JRE functionality was made part of the analysis scope by including the JRE V1.4.2_02 system and extension libraries. To reduce the size of the analysis scope, the MARCO tool was customized to build the analysis scope based on the plug-in dependencies. The number of classes in the analysis scope was still greater than 20,000. Table 1 shows, for some Eclipse plug-ins, how many classes and methods were included in the ARIG, the time employed to run the whole analysis (including the ARIG construction, which on average takes 96% of the total time), the number of nodes and edges in the ARIG, the instructions count, and the number of `doPrivileged()` locations suggested by the tool. On average, 50% of the code portions candidate to become privileged contained malicious tainted variables—an indication that `doPrivileged()` should not be used. The `osgi` plug-in was the only one that already contained calls to `doPrivileged()`. The total number was 29, and 8 of those were unnecessary or redundant.

Other analyses were performed on large commercial products (over 20,000 classes in the analysis scope), based on the JRE V1.1 access-control model. The goal was to identify their privileged-code requirements to allow them to successfully run with the Java 2 access-control model enabled.

7 Related Work

Privileged code has historic roots in the 1970's. The Digital Equipment Corporation (DEC) Virtual Address Extension/Virtual Memory System (VAX/VMS) operating system had a feature similar to the `doPrivileged()` method in Java 2 and the `Assert()` method in CLR. The VAX/VMS feature was called *privileged images*. Privileged images were similar to UNIX `setuid` programs, except that privileged images ran in the same process as all the user's other unprivileged programs. This meant that they were considerably easier to attack than UNIX `setuid` programs because they lacked the usual separate process/separate address space protections. One example of an attack on privileged images is demonstrated in a paper by Koegel, Koegel, Li, and Miruke [23].[6]

More recently, static and dynamic analysis techniques have both been used for modelling authorization algorithms. Much of the work has focused on performance optimizations or on providing alternatives to the existing approaches employed by Java 2 [29, 28] and CLR [16]. Pottier, Skalka, and Smith [30] extend and formalize Wallach's security passing style [41] via type theory using a λ-calculus, called λ_{sec}. However, their approach does not model all of Java's authorization characteristics, including multi-threaded code and analysis of incomplete programs [31], nor does it compute the authorization object, which often includes identifying the `String` parameters to the `Permission` object's constructor. Bartoletti, Degano, and Ferrari [5] are interested in optimizing performance of run-time authorization testing. This is done by eliminating redundant tests and relocating others as is needed. Additionally [6], they investigate ways in which program transformations can preserve security properties in existing code, particularly in the context of Java. Specifically, the transformations they study include redundant authorization tests elimination, dead code elimination, method inlining, and an eager evaluation strategy for stack inspection. While their model takes privileged code into account, they assume that privileged code has already been inserted, and do not solve the problem of detecting which portions of library code should be made privileged. Banerjee and Naumann [4] apply denotational semantics to show the equivalence of *eager* and *lazy* semantics for stack inspection, provide a static analysis of *safety*, and identify transformations that can remove unnecessary authorization tests. Significant limitations to this approach are that the analyses are limited to a single thread, and incomplete-program analyses are not supported. Jensen, Le Métayer, and Thorn [20] focus on proving that code is secure with respect to a global security policy. Their model adopts operational semantics to prove the properties, using a two-level temporal logic, and shows how to detect redundant authorization tests. They assume all of the code is available for analysis, and that a call graph can be constructed for the code. Felten, Wallach, Dean, and Balfanz have studied a number of security problems related to mobile code [39, 12, 41, 8, 40, 10, 9]. In particular, they present a formalization of stack introspection, which examines authorization based on the principals currently active in a thread stack at run time (*security state*). An authorization optimization technique, called *security passing style*, encodes the security state of an application while the application is executing [41]. Each method is modified so that it passes a security token as part of each invocation. The token represents an encoding of the security state at each stack frame, as well as the result of any authorization test encountered. By running the application and encoding the security state, the security passing

[6] In a private communication with Dr. Paul A. Karger [21], he indicated that privileged images had been a very significant source of security attacks in the VAX/VMS operating system, and required many patches and updates over the years. He did extensive work on resolving those problems at DEC in the 1979-1980 timeframe.

style explores subgraphs of the comparable invocation graph, and discovers the associated security states and authorizations. The purpose of their work is to optimize the authorization performance, while ours is to discover which portions of library code should be made privileged. Our approach analyzes all the possible execution paths, even those that may not be discovered by a limited number of test cases. Rather than analyzing security policies as embodied by existing code, Erlingsson and Schneider [14] describe a system that inlines reference monitors into the code to enforce specific security policies. Their objective is to define a security policy and then inject authorization points into the code. This approach can reduce or eliminate redundant authorization tests. Koved, Pistoia, and Kershenbaum [24] describe an algorithm and system for computing Java 2 security authorization requirements for existing Java code. Their algorithm, which is the starting point for this paper, covers many of the subtle aspects of Java 2 security, including authorization requirements for multi-threaded applications and analysis of incomplete programs [31], for the computation of an ARIG.

The notion of tainted variables became known with the Perl language. In Perl, using the `-T` option allows detecting tainted variables [38]. Shankar, Talwar, Foster, and Wagner present a tainted-variable analysis for CQual using constraint graphs [33]. To find format string bugs, CQual uses a type-qualifier system [15] with two qualifiers: *tainted* and *untainted*. The types of values that can be controlled by an untrusted adversary are qualified as being tainted, and the rest of the variables are qualified as untainted. A constraint graph is constructed for a CQual program. If there is a path from a tainted node to an untainted node in the graph, an error is flagged. Newsome and Song [27] propose a dynamic tainted-variable analysis that catches errors by monitoring tainted variables at run time. Data originating or arithmetically derived from untrusted sources, such as the network, are marked as tainted. Tainted variables are tracked at run time, and when they are used in a dangerous way an attack is detected. Volpano, Irvine, and Smith [37] relate tainted-variable analysis to enforcing information flow policies through typing. Ashcraft and Engler [3] also use tainted-variable analysis to detect software attacks due to tainted variables. Their approach provides user-defined sanity checks to untaint potentially tainted variables. In Java 2, Enterprise Edition (J2EE), access rights are defined in terms of operations on components, instead of the data encapsulated and used by the components. Naumovich and Centonze [26] address the need for specifying access rights on data. Access right support for data can simplify sanity checks for tainted variables. For instance, a tainted variable is benign for those clients who have access rights over the data referenced by the tainted variable.

8 Conclusion

In this paper, we presented an interprocedural analysis for safely adding privileged code in order to ensure that no unnecessary access rights are granted to client code, and that tainted variables are not exploited. Our approach for privileged-code and tainted-variable analysis is built on top of an access-rights analysis and uses an Access-Rights Invocation Graph (ARIG). As part of the analysis, we solve a number of other related problems, including identification of unnecessary and redundant privileged code and flagging when tainted variables are benign or malicious. We have implemented the analysis described in this paper and are currently using it to identify security violations due to privileged code in large libraries and applications. Our analysis technique scales well enough to produce usable results on large applications and libraries. While the analysis techniques described in this paper are in the context of Java code, the basic concepts are applicable to privileged-code placement and tainted-variable analysis issues in non-Java-based systems as well.

Acknowledgments

The authors would like to thank their colleagues at the IBM T. J. Watson Research Center: Julian Dolby, Ted Habeck, Paul Karger, Aaron Kershenbaum, Michael Steiner, Sam Weber, and Xiaolan Zhang for their invaluable technical contributions to this paper. Thanks also to the ECOOP 2005 reviewers for their insightful comments and suggestions, which helped to improve this paper.

References

1. Ole Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference Of Parametric Polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 2–26. Springer-Verlag, August 1995.
2. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, January 1986.
3. Ken Ashcraft and Dawson Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 143–159, Oakland, CA, USA, May 2002. IEEE Computer Society.
4. Anindya Banerjee and David A. Naumann. A Simple Semantics and Static Analysis for Java Security. Technical Report CS2001-1, Stevens Institute of Technology, Hoboken, NJ, USA, July 2001.
5. Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. Static Analysis for Stack Inspection. In *Proceedings of International Workshop on Concurrency and Coordination, Electronic Notes in Theoretical Computer Science*, volume 54, Amsterdam, The Netherlands, 2001. Elsevier.
6. Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. Stack Inspection and Secure Program Transformations. *International Journal of Information Security*, 2(3):187–217, August 2004.
7. Frédéric Besson, Tomasz Blanc, Cédric Fournet, and Andrew D. Gordon. From Stack Inspection to Access Control: A Security Analysis for Libraries. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*, pages 61–75, Pacific Grove, CA, USA, June 2004. IEEE Computer Society.
8. Drew Dean. The Security of Static Typing with Dynamic Linking. In *Proceedings of the 4th ACM conference on Computer and Communications Security*, pages 18–27, Zurich, Switzerland, 1997. ACM Press.

9. Drew Dean, Edward W. Felten, and Dan S. Wallach. Java Security: From HotJava to Netscape and beyond. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 190–200, Silver Spring, MD, USA, 1996. IEEE Computer Society Press.
10. Drew Dean, Edward W. Felten, Dan S. Wallach, and Dirk Balfanz. Java Security: Web Browsers and Beyond. Technical Report 566-597, Princeton University, Princeton, NJ, USA, February 1997.
11. Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101, Aarhus, Denmark, August 1995. Springer-Verlag.
12. Richard Drews Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, Princeton, NJ, USA, January 1999.
13. Eclipse Project, `http://www.eclipse.org`.
14. Úlfar Erlingsson and Fred B. Schneider. IRM Enforcement of Java Stack Inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 246–255, Oakland, CA, USA, May 2000. IEEE Computer Society.
15. Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-Sensitive Type Qualifiers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, Berlin, Germany, June 2002.
16. Adam Freeman and Allen Jones. *Programming .NET Security*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, June 2003.
17. Li Gong, Gary Ellison, and Mary Dageforde. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, Reading, MA, USA, second edition, May 2003.
18. George Grätzer. *General Lattice Theory*. Birkhäuser, Boston, MA, USA, second edition, January 2003.
19. Sumit Gulwani and George C. Necula. Path-sensitive Analysis for Linear Arithmetic and Uninterpreted Functions. In *11th Static Analysis Symposium*, volume 3148 of *LNCS*, pages 328–343. Springer-Verlag, August 2004.
20. Thomas P. Jensen, Daniel Le Métayer, and Tommy Thorn. Verification of Control Flow Based Security Properties. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 89–103, Oakland, CA, USA, May 1999.
21. Paul A. Karger, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA. Private communication, 17 December 2004.
22. Gary A. Kildall. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 194–206, Boston, MA, USA, 1973. ACM Press.
23. John F. Koegel, Rhonda M. Koegel, Zhiming Li, and Dattaram T. Miruke. A Security Analysis of VAX VMS. In *ACM '85: Proceedings of the 1985 ACM Annual Conference on the Range of Computing: Mid-80's Perspective*, pages 381–386. ACM Press, 1985.
24. Larry Koved, Marco Pistoia, and Aaron Kershenbaum. Access Rights Analysis for Java. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 359–372, Seattle, WA, USA, November 2002. ACM Press.
25. Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, June 1997.
26. Gleb Naumovich and Paolina Centonze. Static Analysis of Role-Based Access Control in J2EE Applications. *SIGSOFT Software Engineering Notes*, 29(5):1–10, September 2004.

27. James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, San Diego, CA, USA, February 2005. IEEE Computer Society.
28. Marco Pistoia, Nataraj Nagaratnam, Larry Koved, and Anthony Nadalin. *Enterprise Java Security*. Addison-Wesley, Reading, MA, USA, February 2004.
29. Marco Pistoia, Duane Reller, Deepak Gupta, Milind Nagnur, and Ashok K. Ramani. *Java 2 Network Security*. Prentice Hall PTR, Upper Saddle River, NJ, USA, second edition, August 1999.
30. François Pottier, Christian Skalka, and Scott F. Smith. A Systematic Approach to Static Access Control. In *Proceedings of the 10th European Symposium on Programming Languages and Systems*, pages 30–45. Springer-Verlag, 2001.
31. Barbara G. Ryder. Dimensions of Precision in Reference Analysis of Object-Oriented Languages. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 126–137, Warsaw, Poland, April 2003. Invited Paper.
32. Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE*, volume 63, pages 1278–1308, September 1975.
33. Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, Washington, DC, USA, August 2001.
34. Standard Performance Evaluation Corporation Java Business Benchmark 2000 (SPECjbb2000), `http://www.spec.org`.
35. Sun Microsystems, Security Code Guidelines, `http://java.sun.com`.
36. Frank Tip and T. B. Dinesh. A Slicing-based Approach for Locating Type Errors. *ACM Transactions on Software Engineering and Methodology*, 10(1):5–55, 2001.
37. Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2-3):167–187, January 1996.
38. Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, third edition, July 2000.
39. Dan S. Wallach. *A New Approach to Mobile-Code Security*. PhD thesis, Princeton University, Princeton, NJ, USA, January 1999.
40. Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible Security Architectures for Java. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 116–128, Saint Malo, France, 1997. ACM Press.
41. Dan S. Wallach and Edward W. Felten. Understanding Java Stack Inspection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 52–63, Oakland, CA, USA, May 1998.

State Based Ownership, Reentrance, and Encapsulation

Anindya Banerjee[1,⋆] and David A. Naumann[2,⋆⋆]

[1] Kansas State University, Manhattan KS 66506 USA
ab@cis.ksu.edu
[2] Stevens Institute of Technology, Hoboken NJ 07030 USA
naumann@cs.stevens.edu

Abstract. A properly encapsulated data representation can be revised for refactoring or other purposes without affecting the correctness of client programs and extensions of a class. But encapsulation is difficult to achieve in object-oriented programs owing to heap based structures and reentrant callbacks. This paper shows that it is achieved by a discipline using assertions and auxiliary fields to manage invariants and transferrable ownership. The main result is representation independence: a rule for modular proof of equivalence of class implementations.

1 Introduction

You are responsible for a library consisting of many Java classes. While fixing a bug or refactoring some classes, you revise the implementation of a certain class in a way that is intended not to change its observable behavior, e.g., an internal data structure is changed for reasons of performance. You are in no position to check, or even be aware of, the many applications that use the class via its instances or by subclassing it. In principle, the class could have a full functional specification. It would then suffice to prove that the new version meets the specification. In practice, full specifications are rare. Nor is there a well established logic and method for modular reasoning about the code of a class in terms of the specifications of the classes it uses, without regard to their implementations or the users of the class in question [20] (though progress has been made). One problem is that encapsulation, crucial for modular reasoning about invariants, is difficult to achieve in programs that involve shared mutable objects and reentrant callbacks which violate simple layering of abstractions. Yet complicated heap structure and calling patterns are used, in well designed object-oriented programs, precisely for orderly composition of abstractions in terms of other abstractions.

There is an alternative to verification with respect to a specification. One can attempt to prove that the revised version is behaviorally equivalent to the original. Of course their behavior is not identical, but at the level of abstraction of source code (e.g., modulo specific memory addresses), it may be possible to show equivalence of behavior. If any specifications are available they can be taken into account using assert statements.

There is a standard technique for proving equivalence [18, 24]: Define a *coupling relation* to connect the states of the two versions and prove that it has the *simulation*

⋆ Supported in part by NSF grants CCR-0209205, ITR-0326577, and CCR-0296182.
⋆⋆ Supported in part by NSF grants CCR-0208984, CCF-0429894, and by Microsoft Research.

property, i.e., it holds initially and is preserved by parallel execution of the two versions of each method. In most cases, one would want to define a *local coupling* relation for a single pair of instances of the class, as methods act primarily on a target object (self) and the *island* of its representation objects; an *induced coupling* for complete states is then obtained by a general construction. A language with good encapsulation should enjoy an *abstraction* or *representation independence* theorem that says a simulation for the revised class induces a simulation for any program built using the class. Suitable couplings are the identity except inside the abstraction boundary and an *identity extension lemma* says simulation implies behavioral equivalence of two programs that differ only by revision of a class. Again, such reasoning can be invalidated by heap sharing, which violates encapsulation of data, and by callbacks, which violate hierarchical control structure.

There is a close connection between the equivalence problem and verification: verification of object oriented code involves object invariants that constrain the internal state of an instance. Encapsulation involves defining the invariant in a way that protects it from outside interference so it holds globally provided it is preserved by the methods of the class of interest. Simulations are like invariants over two copies of the state space, and again modular reasoning requires that the coupling for a class be independent from outside interference. *The main contribution of this paper is a representation independence theorem using a state-based discipline for heap encapsulation and control of callbacks.*

Extant theories of data abstraction assume, in one way or another, a hierarchy of abstractions such that control does not reenter an encapsulation boundary while already executing inside it. In many programming languages it is impossible to write code that fails to satisfy the assumption. But it is commonplace in object oriented programs for a method *m* acting on some object *o* to invoke a method on some other object which in turn leads to invocation of some method on *o* —possibly *m* itself— while the initial invocation of *m* is in progress. This makes it difficult to reason about when an object's invariant holds [20, 25]; we give an example later.

There is an analogous problem for reasoning with simulations. In previous work [2] we formulated an abstraction theorem that deals with sharing and is sound for programs with reentrant callbacks, but it is not easy to apply in cases where reentrant callbacks are possible. The theorem allows the programmer to assume that all methods preserve the coupling relation when proving simulation, i.e., when reasoning about parallel execution of two versions of a method of the class of interest. This assumption is like verifying a procedure implementation under the assumption that called procedures are correct. But the assumption that called methods preserve the coupling is of no use if the call is made in an uncoupled intermediate state. For the examples in [2], we resort to ad hoc reasoning for examples involving callbacks.

In a recent advance, [6, 21] reentrancy is managed using an explicit auxiliary (or *ghost*) field *inv* to designate states in which an object invariant is to hold. Encapsulation is achieved using a notion of ownership represented by an auxiliary mutable field *own*. This is more flexible than type-based static analyses because the ownership invariant need only hold in certain flagged states. Heap encapsulation is achieved not by disallowing boundary-crossing pointers but by limiting, in a state-dependent way, their use.

Reasoning hinges on a global *program invariant* that holds in all states, using *inv* fields to track which object invariants are temporarily not in force because control is within their encapsulation boundary. When *inv* holds, the object is said to be *packed*; a field may only be updated when the object is unpacked.

In this paper we adapt the *inv*/*own* discipline [6, 21][1] to proving class equivalence by simulation. The *inv* fields make it possible for an induced coupling relation to hold at some pairs of intermediate states during parallel execution of two alternative implementations. This means that the relation-preservation hypothesis of the abstraction theorem can be used at intermediate states even when the local coupling is not in force. So per-method modular reasoning is fully achieved. In large part the discipline is unchanged, as one would hope in keeping with the idea that a coupling is just an invariant over two parallel states. But we have to adapt some features in ways that make sense in terms of informal considerations of information hiding. The discipline imposes no control on field reads, only writes, but for representation independence we need to control reads as well. The discipline also allows ownership transfer quite freely, though it is not trivial to design code that correctly performs transfers. For representation independence, the transfer of previously-encapsulated data to clients (an unusual form of controlled "rep exposure" [16]) is allowed but must occur only in the code of the encapsulating class; even then, it poses a difficult technical challenge. The significance of our adaptations is discussed in Section 7.

A key insight is that, although transferring ownership and packing/unpacking involve only ghost fields that cannot affect program execution, it is useful to consider them to be observable. It is difficult to reason about two versions of a class, in a modular way, if they differ in the way objects cross the encapsulation boundary or in which methods assume the invariant is in force. The requisite similarity can be expressed using assert statements so we can develop a theory based on this insight without the need to require that the class under revision has any specifications.

Contributions. The main contributions are (a) formulation of a notion of instance-based coupling analogous to invariants in the *inv*/*own* discipline; (b) proof of a representation independence theorem for a language with inheritance and dynamic dispatch, recursive methods and callbacks; mutable objects, type casts, and recursive types; and (c) results on identity extension and use of the theorem to prove program equivalence. Together these constitute a rule by which the reasoner considers just the methods of the revised class and concludes that the two versions yield equivalent behavior for any program context.

The theorem allows ownership transfers that cross encapsulation boundaries: from client to abstraction [16], between abstractions, and even from abstraction to client [29, 4]. The theorem supports the most important form of modularity: reasoning about one method implementation (or rather, one corresponding pair) at a time —on the assumption that all methods preserve the coupling (even the one in question, modulo termination). Our theorem also supports local reasoning in the sense that a single instance

[1] Called the "Boogie methodology" in the context of the Spec# project [7] at Microsoft Research, which implements the discipline as part of a comprehensive verification system inspired by the ESC projects.

(or pair of instances) is considered, together with the island comprised of its currently encapsulated representation objects.

The discipline can be used in any verification system that supports ghost variables and assertions. So our formalism treats predicates in assertions semantically, avoiding ties to any particular logic or specification formalism.

Related Work Besides the inv/own Discipline. Representation independence is needed not only for modular proof of equivalence of class implementations but also for modular reasoning about improvements (called data refinement). Such reasoning is needed for correctness preserving refactoring. The refactoring rules of Borba et al. [10] were validated using the data refinement theory of Cavalcanti and Naumann [13] which does not model sharing/aliasing. We plan to use the present result to overcome that limitation. Representation independence has also been used to justify treating a method as pure if none of its side effects are visible outside an encapsulation boundary [8, 26].

Representation independence is proved in [2] for a language with shared mutable objects on the basis of ownership confinement imposed using restrictions expressed in terms of ordinary types; but these restrictions disallow ownership transfer. The results are extended to encompass ownership transfer in [4] but at the cost of substantial technical complications and the need for reachability analysis at transfer points, which are designated by explicit annotations. Like the present paper, our previous results are based on a semantics in which the semantics of primitive commands is given in straightforward operational terms. It is a denotational semantics in that a command denotes a state transformer function, defined by induction on program structure. To handle recursion, method calls are interpreted relative to a *method environment* that gives the semantics of all methods. This is constructed as the limit of approximations, each exact up to a certain maximum calling depth. This model directly matches the recursion rule of Hoare logic, of which the abstraction theorem is in some sense a generalization.

For simple imperative code and single-instance modules, O'Hearn *et al.* [29, 23] have proved strong rules for local reasoning about object invariants and simulations using separation logic which, being state based, admits a notion of ownership transfer.

Confinement disciplines based on static analysis have been given with the objective of encapsulation for modular reasoning, though mostly without formal results on modular reasoning [14, 11]. Work using types makes confinement a *program invariant*, i.e., a property required to hold in every reachable state. This makes it difficult to transfer ownership, due to temporary sharing at intermediate states. Most disciplines preclude transfer (e.g., [15, 11]); where it is allowed, it is achieved using nonstandard constructs such as destructive reads and restrictive linearity constraints (e.g., [12, 30]).

Outline. Sect. 2 sketches the *inv/own* discipline. It also sketches an example of the use of simulation to prove equivalence of two versions of a class involving reentrant callbacks, highlighting the problems and the connection between our solution and the *inv/own* discipline. Sect. 3 formalizes the language for which our result is given and Sect. 4 formalizes the discipline in our semantics. Sect. 5 gives the main definitions—proper annotation, coupling, simulation—and the abstraction theorem. Sect. 6 connects simulation with program equivalence. Sect. 7 discusses future work and assesses our adaptation of the discipline. For lack of space, all proofs are omitted and can be found in the companion technical report, which also treats generics [5].

2 Background and Overview

2.1 The *inv*/*own* Discipline

To illustrate the challenge of reentrant callbacks as well as the state based ownership discipline, we consider a class Queue that maintains a queue of tasks. Each task has an associated limit on the number of times it can be run. Method Queue.runAll runs each task that has not exceeded its limit. For simplicity we refrain from using interfaces; class Task in Fig. 1 serves as the interface for tasks. Class Qnode in the same Figure is used by Queue which maintains a singly linked list of nodes that reference tasks. Field count tracks the number of times the task has been run. For brevity we omit initialization and constructors throughout the examples.

Fig. 2 gives class Queue. One intended invariant of Queue is that no task has been run more times than its limit. This is expressed, in a decentralized way, by the invariant declared in Qnode. Some notation: we write $\mathscr{I}^{Qnode}(o)$ for the predicate o.tsk≠null and o.count≤o.limit.

Another intended invariant of Queue is that runs is the sum of the count fields of the nodes reached from tsks. This is the declared $\mathscr{I}^{Queue}$ of Fig. 2. (The reader may think of other useful invariants, e.g., that the list is null-terminated.) Note that at intermediate points in the body of Queue.runAll, $\mathscr{I}^{Queue}$ does not hold because runs is only updated after the loop. In particular, $\mathscr{I}^{Queue}$ does not hold at the point where p.run() is invoked.

```
class Task {   void run(){ }   }
class Qnode {
  Task tsk;   Qnode nxt;   int count, limit;
  invariant tsk ≠ null ∧ 0≤count≤limit;
  ... // constructor elided (in subsequent figures these ellipses are elided too)
  void run() { tsk.run(); count := count+1; }
  void setTsk(Task t, int lim) {
    tsk := t; limit := lim; count := 0; pack self as Qnode; }   }
```

Fig. 1. Classes Task and Qnode. The pack statement is discussed later

```
class Queue {
  Qnode tsks;   int runs := 0;
  invariant runs = (Σp ∈ tsks.nxt* | p.count);
  int getRuns() { result := runs; }
  void runAll() {
    Qnode p := tsks; int i := 0;
    while p ≠ null do {
      if p.getCount() < p.getLimit() then p.run(); i := i+1; fi; p := p.getNxt(); }
    runs := runs+i; }
  void add(Task t, int lim){
    Qnode n := new Qnode; n.setTsk(t,lim); n.setNxt(tsks); tsks := n; } }
```

Fig. 2. Class Queue

For an example reentrant callback, consider tasks of the following type.

class RTask extends Task { Queue q; **void** run(){q.runAll(); } ...}

Consider a state in which o points to an instance of Queue and the first node in the list, o.tsks, has count=0 and limit=1. Moreover, suppose field q of the first node's task has value o. Invocation of o.runAll diverges: before count is incremented to reflect the first invocation, the task makes a *reentrant call* on o.runAll —in a state where $\mathscr{I}^{Queue}$ does not hold. In fact runAll again invokes run on the first task and the program fails due to unterminating recursion.

As another example, suppose RTask.run is instead **void** run(){q.getRuns();} . This seems harmless, in that getRuns neither depends on $\mathscr{I}^{Queue}$ nor invokes any methods —it is even useful, returning a lower bound on the actual sum of runs. It typifies methods like state readers in the observer pattern, that are intended to be invoked as reentrant callbacks.

The examples illustrate that it is sometimes but not always desirable to allow a reentrant callback when an object's invariant is violated temporarily by an "outer" invocation. The ubiquity of method calls makes it impractical to require an object's invariant to be reestablished before making *any* call —e.g., the point between n.setTsk and n.setNxt of method add in Fig. 2 — although this is sound and has been proposed in the literature on object oriented verification [17, 22].

A better solution is to prevent just the undesirable reentrant calls. One could make the invariant an explicit precondition, e.g., for runAll but not getRuns. This puts responsibility on the caller, e.g., RTask.run cannot establish the precondition and is thus prevented from invoking runAll. But an object invariant like $\mathscr{I}^{Queue}$ involves encapsulated state not suitable to be visible in a public specification.

The solution of the Boogie methodology [6, 21] is to introduce a public ghost field, *inv*, that explicitly represents whether the invariant is in force. In the lingo, *o.inv* says object o is *packed*. Special statements **pack** and **unpack** set and unset *inv*.

A given object is an instance not only of its class but of all its superclasses, each of which may have invariants. The methodology takes this into account as follows. Instead of *inv* being a boolean, as in the simplified explanation above, it ranges over class names C such that C is a superclass of the object's allocated type. That is, it is an invariant (enforced by typing rules) that $o.inv \geq type(o)$ where $type(o)$ is the dynamic type of o. The discipline requires certain assertions preceding pack and unpack statements as well as field updates, to ensure that the following is a *program invariant* (i.e., it holds in all reachable states).

$$o.inv \leq C \Rightarrow \mathscr{I}^C(o) \tag{1}$$

for all C and all allocated objects o. That is, if o is packed at least to class C then the invariant $\mathscr{I}^C$ for C holds. Perhaps the most important stipulated assertion is that $\mathscr{I}^C(o)$ is required as precondition for packing o to level C.

Fig. 3 shows how the discipline is used for class Queue. Assertions impose preconditions on runAll and add which require that the target object is packed to Queue. In runAll, the **unpack** statement sets *inv* to the superclass of Queue, putting the task in a position where it cannot establish the precondition for a reentrant call to runAll, although it can still call getRuns which imposes no precondition on *inv*. After the update

```
void runAll() {    assert self.inv = Queue && ! self.com;
   unpack self from Queue;
   Qnode p := self.tsks; int i := 0;
   while p ≠ null do {
      if p.getCount() < p.getLimit() then p.run(); i := i+1; fi; p := p.getNxt(); }
   self.runs := self.runs + i;
   pack self as Queue; }
void add(Task t, int lim){    assert self.inv = Queue && ! self.com;
   unpack self from Queue;
   Qnode n := new Qnode; setown n to (self,Queue);
   n.setNxt(tsks); n.setTsk(t,lim); self.tsks := n;
   pack self as Queue; } }
```

Fig. 3. Methods of class Queue with selected annotations

Table 1. Stipulated preconditions of field update and of the special commands

assert $e_1.inv > C$;   /* where C is the class that declares f; i.e., $f \in dom(dfields\,C)$ */
$e_1.f{:=}e_2$

assert $e.inv = superC \wedge \mathscr{I}^C(e) \wedge \forall p \mid p.own = (e,C) \Rightarrow \neg p.com \wedge p.inv = type\,p$;
pack e **as** C /* sets $e.inv := C$ and $p.com := true$ for all p with $p.own = (e,C)$ */

assert $e.inv = C \wedge \neg e.com$;
unpack e **from** C /* sets $e.inv := superC$ and $p.com := false$ for all p with $p.own = (e,C)$ */

assert $e_1.inv =$ Object $\wedge (e_2 = \textbf{null} \vee e_2.inv > C)$;
setown e_1 **to** (e_2,C) /* sets $e_1.own := (e_2,C)$ */

to runs, $\mathscr{I}^{Queue}$ holds again as required by the precondition (not shown) of **pack**. The ghost field *com* is discussed below.

In order to maintain (1) as a program invariant, it is necessary to control updates to fields on which invariants depend. The idea is that, to update field f of some object p, all objects o whose invariant depends on $p.f$ must be unpacked. Put differently, $\mathscr{I}(o)$ should depend only on state encapsulated for o. The discipline uses a form of ownership for this purpose: $\mathscr{I}(o)$ may depend only on objects transitively owned by o. For example, an instance of Queue owns the Qnodes reached from field tsks.

Ownership is embodied in an auxiliary field *own*, so that if $p.own = (o,C)$ then o directly owns p and an admissible invariant $\mathscr{I}^D(o)$ may depend on p for types D with $type(o) \leq D \leq C$. The objects transitively owned by o are called its *island*. For modular reasoning, it is not feasible to require as an explicit precondition for each field update that all transitive owners are unpacked. A third ghost field, *com*, is used to enforce a protocol whereby packing/unpacking is dynamically nested or bracketed (though this need not be textually apparent).

In addition to (1), two additional conditions are imposed as program invariants, i.e., to hold in all reachable states of all objects. The first may be read "an object is committed to its owner if its owner is packed". The second says that a committed object is fully packed. These make it possible for an assignment to $p.f$ to be subject only to the precondition $p.inv > C$ where C is the class that declares f.

The invariants are formalized in Def. 3 in Sect. 4. The stipulated preconditions appear in Table 1, which also describes the semantics of the pack and unpack statements in detail.[2] The diligent reader may enjoy completing the annotation of Fig. 3 according to the rules of Table 1. Consult [6, 21] for more leisurely introductions to the discipline.

2.2 Representation Independence

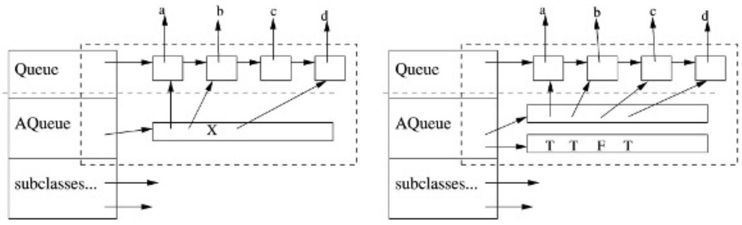
Consider the subclass AQueue of Queue declared in Fig. 4. It maintains an array, actsks, of tasks which is used in an overriding declaration of runAll intended as an optimization for the situation where many tasks are inactive (have reached their limit). We've dropped runs and getRuns for brevity. Method add exhibits a typical pattern: unpack to establish the condition in which a super call can be made (since the superclass unpacks from its own level); after that call, reestablish the current class invariant. (This imposes proof obligations on inheritance, see [6].)

The implementation of Fig. 4 does not set actsks[i] to null immediately when the task's count reaches its limit; rather, that situation is detected on the subsequent invocation of runAll. An alternative implementation is given in Fig. 5; it uses a different data structure and handles the limit being reached as soon as it happens. Both implementations maintain an array of Qnode, but in the alternative implementation, its array artsk is accompanied by a boolean array brtsk. Instead of setting entry i null when the node's task has reached its limit, brtsk[i] is set false.

We claim that the two versions are equivalent, in the context of arbitrary client programs (and subclasses, though for lack of space we do not focus on subclasses in the sequel). We would like to argue as follows. Let $filt1(o.\mathsf{actsks})$ be the sequence of non-null elements of $o.\mathsf{actsks}$ with $\mathsf{count} < \mathsf{limit}$. Let $filt2(ts, bs)$ take an array ts of tasks and a same-length array bs of booleans and return the subsequence of those tasks n in ts where bs is true and $n.\mathsf{count} < n.\mathsf{limit}$. Consider the following relation that connects a state for an instance o of the original implementation (Table 4) with an instance o' for the alternative: $filt1(o.\mathsf{actsks}) = filt2(o'.\mathsf{artsk}, o'.\mathsf{brtsk})$. The idea is that methods of the new version behave the same as the old version, modulo this change of representation. That is, for each method of AQueue, parallel execution of the two versions from a related pair of states results in a related pair of outcomes. (For this to hold we need to conjoin to the relation the invariants associated with the two versions, e.g., the second version requires $\mathsf{artsk.length{=}brtsk.length}$.)

The left side of the picture below is an instance of some subclass of AQueue, sliced into the fields of Queue, AQueue, and subclasses; dashed lines show the objects encapsulated at the two levels relevant to reasoning about AQueue —namely the Qnodes reached from tsks and the array actsks.

[2] We omit the preconditions $e \neq \textbf{null}$ and "e not error" that are needed for the rest of the precondition to be meaningful. Different verification systems make different choices in handling errors in assertions. Our formulation follows [28] and differs superficially from [6, 21].



On the right is an instance for the alternate implementation of AQueue. It is the connection between these two islands that is of interest to the programmer. The “a”. . . “d” of the figure indicate that both versions reference the same sequence of tasks, although those tasks are not part of the islands.

In general, a *local coupling* is a binary relation on islands. It relates the state of an island for one implementation of the class of interest with an island for the alternative.

A local coupling gives rise to an *induced coupling* relation on the complete program state: Two heaps are related by the induced coupling provided that (a) they can be partitioned into islands and (b) the islands can be put into correspondence so that each corresponding pair is related by the local coupling. Moreover, the remaining objects (not in an island) are related by equality. (More precisely, equality modulo a bijection on locations, to take into account differences in allocation between the two versions.) The details are not obvious and are formalized later.

The point of the abstraction theorem is to justify that it is sufficient to check that the induced coupling is preserved by methods of AQueue, assuming the changed data structure is encapsulated and can neither affect nor be affected by client programs. At first glance one might expect the proof obligation to be that each method of AQueue preserves the local coupling, and indeed this will be the focus of reasoning in practice. But in general a method may act on more than just the island for self, e.g., by invoking

```
class AQueue extends Queue {
  private Qnode[ ] actsks;    private int  alen;
  void  add(Task t, int  lim) {    assert self.inv= AQueue && ! self.com;
    unpack self from AQueue;
    super.add(t,lim); actsks[alen] := self.tsks; self.alen := self.alen+1;
    pack self as AQueue; }
  void  runAll() {    assert self.inv= AQueue && ! self.com;
    unpack self from AQueue;
    int  i := self.alen - 1;
    while i ≥ 0 do  {
      Qnode qn := self.actsks[i];
      if  qn ≠ null then  if  qn.getCount() < qn.getLimit()
                          then  qn.run(); else  self.actsks[i] := null; fi; fi;
      i := i - 1; }
    pack self as AQueue; } }
```

Fig. 4. First version of Class AQueue. An invariant: actsks[0..alen-1] contains any n in tsks with n.count $<$ n.limit, in reverse order. (There may also be nulls and some n with n.count $=$ n.limit). The elided constructor allocates actsks and we ignore the issue of the array becoming full

```
class AQueue extends Queue {
  private Qnode[ ] artsk;
  private boolean[ ] brtsk;
  private int len;
  void add(Task t, int lim) {
    assert self.inv= AQueue && ! self.com;
    unpack self from AQueue;
    super.add(t,lim); self.artsk[alen] := self.tsks; self.brtsk[len] := true; self.len := len+1;
    pack self as AQueue; }
  void runAll() {
    assert self.inv= AQueue && ! self.com;
    unpack self from AQueue;
    int i := self.len - 1;
    while i ≥ 0 do {
      if self.brtsk[i] then Qnode n := self.artsk[i]; int diff := n.limit - n.count;
                            if diff ≤ 1 then self.brtks[i] := false; fi;
                            if diff ≠ 0 then n.run(); fi; fi;
      i := i - 1; }
    pack self as AQueue; } }
```

Fig. 5. Alternative implementation of AQueue

methods on client objects or on other instances of AQueue. So the proof obligation is formalized in terms of the induced coupling.

In fact the proof obligation is not simply that each corresponding pair of method implementations preserves the coupling, but rather that they preserve the coupling *under the assumption that any method they invoke preserves the coupling*.[3] There is also a proof obligation for initialization but it is straightforward so we do not discuss it in connection with the examples.

For example, in the case of method runAll, one must prove that the implementations given in Fig. 4 and in Fig. 5 preserve the coupling on the assumption that the invoked methods getCount, getLimit, Qnode.run, etc. preserve the coupling. The assumption is not so important for getCount or getLimit. For one thing, it is possible to fully describe their simple behavior. For another, the alternative implementation of runAll does not even invoke these methods but rather accesses the fields directly.

The assumption about Qnode.run is crucial, however. Because run invokes, in turn, Task.run, essentially nothing is known about its behavior. For this reason both implementations of runAll invoke run on the same tasks in the same order; otherwise, it is hard to imagine how equivalence of the implementations could be verified in a modular way, i.e., reasoning only about class AQueue. But here we encounter the problem with simulation based reasoning that is analogous to the problem with invariants and reentrant callbacks. There is no reason for the coupling to hold at intermediate points of the methods of AQueue. If a method is invoked at such a point, the assumption that the

[3] The reason this is sound is similar to the justification for proof rules for recursive procedures: it is essentially the induction step for a proof by induction on the maximum depth of the method call stack.

called method preserves the coupling is of no use —just as the assumption of invariant-preservation is of no use if a method is invoked in a state where the invariant does not hold.

The Boogie discipline solves the invariant problem for an object o by replacing the declared invariant $\mathscr{I}(o)$ with an implication —see (1)— that is true in all states. As with invariants, so too with couplings: It does not make sense to ask a coupling to hold in every state, because two different implementations with nontrivial differences do not have lockstep correspondence of states. (For example, imagine that in the alternative version, the arrays are compressed every 100th invocation of runAll.) Our generalization of the Boogie idea is that the local coupling relation for a particular (pair of) island(s) is conditioned on an *inv* field so that the local coupling may hold in *some pairs of states* at intermediate points —in particular, at method calls that can lead to reentrant callbacks.

Consider corresponding instances o, o' of the two versions of AQueue. The local coupling serves to describe the corresponding pair of islands when o and o' are packed. So the induced coupling relation on program states requires corresponding pairs of islands to satisfy the local coupling just when they are packed. Because *inv* is part of the behavior observable at the level of reasoning, we can assume both versions follow the same pattern of packing (though not necessarily of control structure) and thus include $o.inv = o'.inv$ as a conjunct of the induced coupling.

Consider the two implementations of runAll. To a first approximation, what matters is that each updates some internal state and then both reach a point where run is invoked. At that point, the *local* coupling does not hold —but the *induced* coupling relation can and does hold, because the island is unpacked. This parallels the way $\mathscr{I}^C(o)$ can be false while $o.inv \leq C \Rightarrow \mathscr{I}^C(o)$ remains true, recall (1). So we can use the assumption about called methods to conclude that the coupling holds after the corresponding calls to run.

The hardest part of the proof for runAll is at the point where the two implementations pack self to *AQueue*. Just as both implementations invoke run (and on the same queue nodes), both need to pack in order to preserve the coupling. And at this point we have to argue that the local coupling is reestablished. To do so, we need to know the state of the internal structures that have been modified. We would like to argue that the only modifications are only those explicit in the code of runAll, but what about the effect of run? Owing to the preconditions on add and runAll, the only possible reentrant callbacks are to getRuns and this does no updates. (In other examples, modifies specifications would be needed at this point for modular reasoning.)

This concludes the sketch of how our abstraction theorem handles reentrant callbacks and encapsulation using the *inv*/*own* discipline. Several features of the discipline need to be adapted, in ways which also make sense in terms of informal considerations of information hiding. The additional restrictions are formalized in Section 5 and their significance discussed in Section 7. As a preview we make the following remarks, using "*Abs*" as the generic name for a class for which two versions are considered.

The discipline does not constrain field access, as reading cannot falsify an invariant predicate. Of course for reasons of information hiding one expects that visibility and alias confinement are used to prevent most or all reads of encapsulated objects. Information hiding is exactly what is formalized by representation independence and indeed the abstraction theorem fails if a client can read fields of encapsulated objects. So every field access $e.f$ is subject to a precondition: If e is transitively owned by some instance o of the class, *Abs*, under revision, then either self is o or else self is transitively owned by o.

Another problematic feature is that "**pack** e **as** C" can occur in any class, so long as its preconditions are established. This means that, unlike traditional theories, an invariant is not simply established at initialization. In our theory the local coupling must be established preceding each "**pack** e **as** *Abs*". We aim for modular reasoning where only *Abs* needs to be considered, so we insist that **pack** e **as** C with $C = Abs$ occurs only in code of *Abs*.

Although the discipline supports hierarchical ownership, our technical treatment benefits from heap partitioning ideas from separation logic (we highlight the connections where possible, e.g., in Proposition 1). For this reason and a more technical one, it is convenient to prevent an instance of *Abs* from transitively owning another instance of *Abs* (lest their islands be nested). This can be achieved by a simple syntactic restriction. It does not preclude that, say, class AQueue can hold tasks that own AQueue objects, because an instance of AQueue owns its representation objects (the Qnodes), not the tasks they contain. Nor does it preclude hierarchical ownership, e.g., *Abs* could own a hashtable that in turn owns some arrays.

Finally, consider ownership transfer across the encapsulation boundary. The hardest case is where a hitherto-encapsulated object is released to a client, e.g., when a memory manager allocates nodes from a free list [29, 4]. This can be seen as a deliberate exposure of representation and thus is observable behavior that must be retained in a revised version of the abstraction. Yet encapsulated data of the two versions can be in general quite different. To support modular reasoning about the two versions, it appears essential to restrict outward transfer of objects encapsulated for *Abs* to occur only in code of *Abs*, where the reasoner can show that the coupling is preserved.

3 An Illustrative Language

Following [6, 21], we formalize the inv/own discipline in terms of a language in which fields have public visibility, to illuminate the conditions necessary for sound reasoning about invariants and simulations. In practice, private and protected visibility and perhaps lightweight alias control would serve to automatically check most of the conditions. This section formalizes the language, adapting notations and typing rules from Featherweight Java [19] and imperative features and the special commands from our previous papers [2, 28].

A complete program is given as a *class table*, CT, that maps class name C to a declaration $CT(C)$ of the form **class** C **extends** D $\{\ \bar{T}\ \bar{f};\ \bar{M}\ \}$. The categories T, M are given by the grammar in Table 2. Barred identifiers like $\bar{T}$ indicate finite lists, e.g., $\bar{T}\ \bar{f}$ stands for a list $\bar{f}$ of field names with corresponding types $\bar{T}$.

Well formed class tables are characterized using typing rules which are expressed using some auxiliary functions that in turn depend on the class table, allowing classes to make mutually recursive references to other classes, without restriction. In particular, this allows recursive methods (so we omit loops). For a class C, $fields(C)$ is defined as the inherited and declared fields of C; $dfields(C)$ is the fields declared in C; $super(C)$ is the direct superclass of C. For a method declaration, $T\ m(\bar{T}_1\ \bar{x})\ \{S\}$ in C, the method type $mtype(m,C)$ is $\bar{T}_1 \to T$ and parameter names, $pars(m,C)$, is $\bar{x}$. For m inherited

Table 2. Grammar

$C \in \mathit{ClassName} \quad m \in \mathit{MethName} \quad f \in \mathit{FieldName} \quad x, \mathsf{self}, \mathsf{result} \in \mathit{VarName}$

$T ::= \textbf{bool} \mid \textbf{void} \mid C$	data type
$M ::= T\ m(\bar{T}\ \bar{x})\ \{S\}$	method declaration
$S ::= x{:=}e \mid e.f{:=}e$	assign to local var. or param., update field
$\mid\ x{:=}\,\textbf{new}\ C \mid x{:=}e.m(\bar{e})$	object creation, method call
$\mid\ T\ x{:=}e\ \textbf{in}\ S \mid S;\ S \mid \textbf{if}\ e\ \textbf{then}\ S\ \textbf{else}\ S\ \textbf{fi}$	local variable, sequence, conditional
$\mid\ \textbf{pack}\ e\ \textbf{as}\ C \mid \textbf{unpack}\ e\ \textbf{from}\ C$	set *inv* to C, set *inv* to *super* C
$\mid\ \textbf{setown}\ e\ \textbf{to}\ (e',C)$	set $e.own$ to (e',C)
$\mid\ \textbf{assert}\ \mathscr{P}$	assert (semantic predicate $\mathscr{P}$)
$e ::= x \mid \textbf{null} \mid \textbf{true} \mid \textbf{false}$	variable, constant
$\mid\ e.f \mid e = e \mid e\ \textbf{is}\ C \mid (C)\ e$	field access, ptr. equality, type test, cast

Table 3. Typing rules for selected expressions and commands

$$\frac{\Gamma \vdash e : C \qquad (f:T) \in \mathit{fields}(C)}{\Gamma \vdash e.f : T} \qquad \frac{\Gamma \vdash e_1 : D_1 \qquad \Gamma \vdash e_2 : D_2 \qquad D_2 \leq C}{\Gamma \vdash \textbf{setown}\ e_1\ \textbf{to}\ (e_2, C)}$$

$$\frac{\Gamma \vdash e : D \quad D \leq C}{\Gamma \vdash \textbf{pack}\ e\ \textbf{as}\ C} \qquad \frac{\Gamma \vdash e : D \quad D \leq C}{\Gamma \vdash \textbf{unpack}\ e\ \textbf{from}\ C}$$

$$\frac{\Gamma \vdash e : D \qquad \mathit{mtype}(m,D) = \bar{T} \to U \qquad x \neq \mathsf{self} \qquad \Gamma \vdash \bar{e} : \bar{U} \qquad \bar{U} \leq \bar{T} \qquad U \leq \Gamma x}{\Gamma \vdash x{:=}e.m(\bar{e})}$$

in C, $\mathit{mtype}(m,C) = \mathit{mtype}(m,D)$ and $\mathit{pars}(m,C) = \mathit{pars}(m,D)$ where D is the direct superclass of C.

For use in the semantics, $\mathit{xfields}(C)$ extends $\mathit{fields}(C)$ by assigning "types" to the auxiliary fields: $\mathit{com} : \textbf{bool}$, $\mathit{own} : \mathsf{owntyp}$, and $\mathit{inv} : (\mathsf{invtyp}\,C)$. (These are not included in *FieldName*.) Neither $\mathsf{invtyp}\,C$ nor owntyp are types in the programming language but there are corresponding semantic domains and the slight notational abuse is convenient.

A *typing context* Γ is a finite function from variable names to types, such that $\mathsf{self} \in \mathit{dom}\,\Gamma$. Selected typing rules for expressions and commands are given in Table 3. A judgement of the form $\Gamma \vdash e : T$ says that expression e has type T in the context of a method of class $\Gamma\,\mathsf{self}$, with parameters and local variables declared by Γ. A judgement $\Gamma \vdash S$ says that S is a command in the same context. A class table CT is well formed if each method declaration $M \in CT(C)$ is well formed in C; this is written $C \vdash M$ and defined by the following rule:

$$\frac{\begin{array}{c}\bar{x} : \bar{T}, \mathsf{self} : C, \mathsf{result} : T \vdash S \\ \text{if } \mathit{mtype}(m, \mathit{super}\,C) \text{ is defined then } \mathit{mtype}(m, \mathit{super}\,C) = \bar{T} \to T \text{ and } \mathit{pars}(m, \mathit{super}\,C) = \bar{x}\end{array}}{C \vdash T\ m(\bar{T}\ \bar{x})\{S\}}$$

To formalize assertions, we prefer to avoid both the commitment to a particular formula language and the complication of an environment for declaring predicate names to be interpreted in the semantics. So we indulge in a mild and commonplace abuse of notation: the syntax of **assert** uses a semantic predicate. We say $\Gamma \vdash \textbf{assert}\ \mathscr{P}$ is well

Table 4. Semantic categories θ and domains $\llbracket\theta\rrbracket$. (Readers familiar with notation for dependent function spaces might prefer to write $\llbracket\mathsf{pre\text{-}heap}\rrbracket = (o : Loc \rightharpoonup \llbracket\mathsf{state}\,(type\ o)\rrbracket)$ and similarly for $\llbracket\mathsf{state}\,C\rrbracket$ and $\llbracket\Gamma\rrbracket$.)

$\theta ::= T \mid \Gamma \mid \theta_\perp$	
$\mid\ \mathsf{owntyp} \mid \mathsf{invtyp}\,C \mid \mathsf{state}\,C$	*own* and *inv* val., object state
$\mid\ \mathsf{pre\text{-}heap} \mid \mathsf{heap} \mid \mathsf{heap}\otimes\Gamma \mid \mathsf{heap}\otimes T$	heap fragment, closed heap, state, result
$\mid\ (\Gamma\vdash\mathsf{cmd}) \mid (\Gamma\vdash T) \mid (C,\bar{x},\bar{T}\to T_1) \mid \mathsf{menv}$	command, expr., method, method envir.

$\llbracket C\rrbracket = \{nil\}\cup\{o\in Loc \mid type\,o \le C\}$ $\qquad\llbracket\mathbf{bool}\rrbracket = \{true, false\}$ $\qquad\llbracket\mathbf{void}\rrbracket = \{it\}$

$\llbracket\mathsf{invtyp}\,C\rrbracket$	$= \{B \mid C\le B\}$
$\llbracket\mathsf{owntyp}\rrbracket$	$= \{(o,C) \mid o = nil \vee type\,o\le C\}$
$\llbracket\mathsf{state}\,C\rrbracket$	$= \{s \mid dom\,s = dom(xfields\,C) \wedge \forall (f:T)\in xfields\,C \mid s\,f\in\llbracket T\rrbracket\}$
$\llbracket\mathsf{pre\text{-}heap}\rrbracket$	$= \{h \mid dom\ h\subseteq_{fin} Loc \wedge \forall o\in dom\ h \mid h\ o\in\llbracket\mathsf{state}\,(type\ o)\rrbracket\}$
$\llbracket\mathsf{heap}\rrbracket$	$= \{h \mid h\in\llbracket\mathsf{pre\text{-}heap}\rrbracket \wedge \forall s\in rng\ h \mid rng\ s\cap Loc\subseteq dom\ h\}$
$\llbracket\Gamma\rrbracket$	$= \{s \mid dom\,s = dom\,\Gamma \wedge s\,\mathsf{self}\neq nil \wedge \forall x\in dom\,s \mid s\,x\in\llbracket\Gamma\,x\rrbracket\}$
$\llbracket\mathsf{heap}\otimes\Gamma\rrbracket$	$= \{(h,s) \mid h\in\llbracket\mathsf{heap}\rrbracket \wedge s\in\llbracket\Gamma\rrbracket \wedge rng\ s\cap Loc\subseteq dom\ h\}$
$\llbracket\mathsf{heap}\otimes T\rrbracket$	$= \{(h,v) \mid h\in\llbracket\mathsf{heap}\rrbracket \wedge v\in\llbracket T\rrbracket \wedge (v\in Loc\Rightarrow v\in dom\ h)\}$
$\llbracket\Gamma\vdash\mathsf{cmd}\rrbracket$	$= \llbracket\mathsf{heap}\otimes\Gamma\rrbracket\to\llbracket(\mathsf{heap}\otimes\Gamma)_\perp\rrbracket$
$\llbracket\Gamma\vdash T\rrbracket$	$= \{v \mid v\in(\llbracket\mathsf{heap}\otimes\Gamma\rrbracket\to\llbracket T\rrbracket_\perp) \wedge \forall h,s \mid v(h,s)\in Loc\Rightarrow v(h,s)\in dom\,h\}$
$\llbracket(C,\bar{x},\bar{T}\to T_1)\rrbracket$	$= \llbracket\mathsf{heap}\otimes(\bar{x}:\bar{T},\mathsf{self}:C)\rrbracket\to\llbracket(\mathsf{heap}\otimes T_1)_\perp\rrbracket$
$\llbracket\mathsf{menv}\rrbracket$	$= \{\mu \mid \forall C,m \mid \mu C m$ is defined iff $mtype(m,C)$ is defined, and $\mu C m\in\llbracket C, pars(m,C), mtype(m,C)\rrbracket$ if $\mu C m$ defined $\}$

formed provided that $\mathscr{P}$ is a set of program states for context Γ. This treatment of assertions is also convenient for taking advantage of a theorem prover's native logic.

Semantics. Some semantic domains correspond directly to the syntax. For example, each data type T denotes a set $\llbracket T\rrbracket$ of values. The meaning of context Γ is a set $\llbracket\Gamma\rrbracket$ of stores; a *store* $s\in\llbracket\Gamma\rrbracket$ is a type-respecting assignment of locations and primitive values to the local variables and parameters given by a typing context Γ. The semantics, and later the coupling relation, is structured in terms of category names θ given in Table 4 which also defines the semantic domains.

A *program state* for context Γ is a pair (h,s) where s is in $\llbracket\Gamma\rrbracket$ and h is a *heap*, i.e., a finite partial function from locations to object states. An *object state* is a type-respecting mapping of field names to values. A command typable in Γ denotes a function mapping each program state (h,s) either to a final state (h_0,s_0) or to the distinguished value $\perp$ which represents runtime errors, divergence, and assertion failure. An *object state* is a mapping from (extended) field names to values. A *pre-heap* is like a heap except for possibly having dangling references. If h,h' are pre-heaps with disjoint domains then we write $h*h'$ for their union; otherwise $h*h'$ is undefined. Function application associates to the left, so $h\,o\,f$ is the value of field f of the object $h\,o$ at location o. We also write $h\,o.f$. Application binds more tightly than binary operator symbols and ",".

We assume that a countable set *Loc* is given, along with a distinguished value *nil* not in *Loc*. We assume given a function *type* from *Loc* to non-primitive types distinct from Object, such that for each C there are infinitely many locations o with $type\ o = C$. This is used in a way that is equivalent to tagging object states with their type.

Table 5. Semantics of selected expressions and commands. To streamline the treatment of $\perp$, the metalanguage expression "let $\alpha = \beta$ in ..." denotes $\perp$ if β is $\perp$. We use function extension notation $[h \mid o \mapsto st]$ for h extended or overridden at o with value st. For brevity the nested function extension for field update is written $[h \mid o.f \mapsto v]$

$$
\begin{array}{ll}
[\![\Gamma \vdash e.f : T]\!](h,s) & = \mathsf{let}\ o = [\![\Gamma \vdash e : C]\!](h,s)\ \mathsf{in\ if}\ o = nil\ \mathsf{then}\ \perp\ \mathsf{else}\ h\,o.f \\
[\![\Gamma \vdash x{:=}e.m(\bar{e})]\!]\mu(h,s) & = \mathsf{let}\ o = [\![\Gamma \vdash e : T]\!](h,s)\ \mathsf{in\ if}\ o = nil\ \mathsf{then}\ \perp\ \mathsf{else} \\
 & \quad \mathsf{let}\ \bar{v} = [\![\Gamma \vdash \bar{e} : \bar{U}]\!](h,s)\ \mathsf{in\ let}\ \bar{x} = pars(m,T)\ \mathsf{in} \\
 & \quad \mathsf{let}\ s_1 = [\bar{x} \mapsto \bar{v}, \mathsf{self} \mapsto o]\ \mathsf{in} \\
 & \quad \mathsf{let}\ (h_1, v_1) = \mu(type\, o)m(h, s_1)\ \mathsf{in}\ (h_1, [s \mid x \mapsto v_1]) \\
[\![\Gamma \vdash \mathbf{assert}\ \mathscr{P}]\!]\mu(h,s) & = \mathsf{if}\ (h,s) \in \mathscr{P}\ \mathsf{then}\ (h,s)\ \mathsf{else}\ \perp
\end{array}
$$

$[\![\Gamma \vdash \mathbf{pack}\ e\ \mathbf{as}\ C]\!]\mu(h,s) =$

$\quad \mathsf{let}\ q = [\![\Gamma \vdash e : D]\!](h,s)\ \mathsf{in\ if}\ q = nil\ \mathsf{then}\ \perp\ \mathsf{else}$

$\quad \mathsf{let}\ h_1 = \lambda p \in dom\ h \mid \mathsf{if}\ h\,p.own = (q,C)\ \mathsf{then}\ [h\,p \mid com \mapsto true]\ \mathsf{else}\ h\,p\ \mathsf{in}\ ([h_1 \mid q.inv \mapsto C],\ s)$

$[\![\Gamma \vdash \mathbf{unpack}\ e\ \mathbf{from}\ C]\!]\mu(h,s) =$

$\quad \mathsf{let}\ q = [\![\Gamma \vdash e : N]\!](h,s)\ \mathsf{in\ if}\ q = nil\ \mathsf{then}\ \perp\ \mathsf{else}$

$\quad \mathsf{let}\ h_1 = \lambda p \in dom\ h \mid \mathsf{if}\ h\,p.own = (q,C)\ \mathsf{then}\ [h\,p \mid com \mapsto false]\ \mathsf{else}\ h\,p\ \mathsf{in}$

$\quad ([h_1 \mid q.inv \mapsto super C],\ s)$

$[\![\Gamma \vdash \mathbf{setown}\ e_1\ \mathbf{to}\ (e_2, C)]\!]\mu(h,s) =$

$\quad \mathsf{let}\ q = [\![\Gamma \vdash e_1 : N_1]\!](h,s)\ \mathsf{in\ if}\ q = nil\ \mathsf{then}\ \perp\ \mathsf{else}$

$\quad \mathsf{let}\ p = [\![\Gamma \vdash e_2 : N_2]\!](h,s)\ \mathsf{in}\ ([h \mid q.own \mapsto (p,C)],\ s)$

The meaning of a derivable command typing $\Gamma \vdash S$ will be defined to be a function sending each method environment μ to an element of $[\![\Gamma \vdash \mathsf{cmd}]\!]$. (The keyword "cmd" just provides notation for command meanings.) That is, $[\![\Gamma \vdash S]\!]\mu$ is a state transformer $[\![\mathsf{heap} \otimes \Gamma]\!] \to [\![(\mathsf{heap} \otimes \Gamma)_\perp]\!]$. The method environment is used only to interpret the method call command. Meanings for expressions and commands are defined, in Table 5, by recursion on typing derivation. The semantics is defined for an arbitrary location-valued function *fresh* such that $type(fresh(C,h)) = C$ and $fresh(C,h) \notin dom\, h$.

The meaning of a well typed method declaration M, of the form $M = T\ m(\bar{T}\ \bar{x})\{S\}$, is the total function in $[\![\mathsf{menv}]\!] \to [\![(C, \bar{x}, \bar{T} \to T)]\!]$ defined as follows: Given a method environment μ, a heap h and a store $s \in [\![\bar{x} : \bar{T}, \mathsf{result} : C]\!]$, first execute S to obtain the updated heap h_0 and the updated store s_0; then return $(h_0, s_0(\mathsf{result}))$. A method environment μ maps each C, m to a meaning obtained in this way or by inheritance. For well formed class table CT, the semantics $[\![CT]\!]$ is defined as the least upper bound of an ascending chain of method environments—the approximation chain—with method declarations interpreted as above and a suitable interpretation for inherited methods. Details omitted.

A *predicate* for state type Γ is just a subset $\mathscr{P} \subseteq [\![\mathsf{heap} \otimes \Gamma]\!]$. For emphasis we can write $(h,s) \models \mathscr{P}$ for $(h,s) \in \mathscr{P}$. Note that $\perp \notin \mathscr{P}$. We give no formal syntax to denote predicates but rather use informal metalanguage for which the correspondence should be clear. For example, "$\mathsf{self}.f \neq$ **null**" denotes the set of (h,s) with $h(s\,\mathsf{self}).f \neq nil$. and "$\forall o \mid \mathscr{P}(o)$" denotes the set of (h,s) such that $(h,s) \models \mathscr{P}(o)$ for all $o \in dom\ h$. Note that quantification over objects (e.g., in Table 1 and Def. 3) is interpreted to mean quantification over allocated locations; the range of quantification can include unreachable objects but this causes no problems.

By contrast with [6, 21], we have taken care to separate the annotations required by the *inv/own* discipline from the semantics of commands. The invariants encoded in the semantic domains (e.g., the value in a field has its declared type and there are no dangling pointers) depend in no way on assertions, only on typing. A similar semantic model has been machine checked in PVS [27].

4 The *inv/own* Discipline

The discipline reviewed in Sect. 2.1 is designed to make (1) a program invariant for every object. This is achieved using additional program invariants that govern ownership. We formalize this as a global predicate, *disciplined*, defined in three steps.

Definition 1 (transitive C- and $C{\uparrow}$-ownership). For any heap h, the relation $o \succ_C^h p$ on $dom\, h$, read "*o owns p at C in h*", holds iff either $(o,C) = h\,p.own$ or there are q and D such that $(o,C) = h\,q.own$ and $q \succ_D^h p$. The relation $o \succ_{C\uparrow}^h p$ holds iff there is some D with $C \leq D$ and $o \succ_D^h p$.

Definition 2 (admissible invariant). A predicate $\mathscr{P} \subseteq [\![\mathsf{heap} \otimes (\mathsf{self} : C)]\!]$ is *admissible as an invariant for C* provided that it is not falsifiable by creation of new objects and for every (h,s) and o,f such that $\mathscr{P}$ depends on $o.f$ in (h,s), field f is neither *inv* nor *com*, and one of the following conditions holds: $o = s(\mathsf{self})$ and f is in $dom(xfields\, C)$ or $s(\mathsf{self}) \succ_{C\uparrow}^h o$.

For dependence on fields of self, the typing condition, $f \in dom(xfields\, C)$, prevents an invariant for C from depending on fields declared in a subclass of C (which could be expressed in a formula using a cast). An invariant can depend on any fields of objects owned at C or above. We refrain from introducing syntax for declaring invariants. In the subsequent definitions, an admissible invariant $\mathscr{I}^C$ is assumed given for every class C. We assume $\mathscr{I}^{Object} = \mathbf{true}$.

Definition 3 (disciplined, $\mathscr{J}$). A heap h is *disciplined* if $h \models \mathscr{J}$ where $\mathscr{J}$ is defined to be the conjunction of the following: $\forall o, C \mid o.inv \leq C \Rightarrow \mathscr{I}^C(o)$

$$\forall o, C, p \mid o.inv \leq C \wedge p.own = (o,C) \Rightarrow p.com$$
$$\forall o \mid o.com \Rightarrow o.inv = type(o)$$

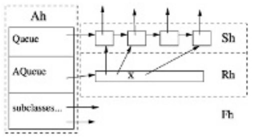
A state (h,s) is *disciplined* if h is. Method environment μ *is disciplined* provided that every method maintains $\mathscr{J}$ (i.e., for any C, m, h, s, if $h \in \mathscr{J}$ and $\mu\, C\, m\,(h,s) = (h_0, v)$—and thus $\mu\, C\, m\,(h,s) \neq \bot$— then $h_0 \in \mathscr{J}$).

Lemma 1 (transitive ownership). Suppose h is disciplined and $o \succ_C^h p$. Then (a) $type\, o \leq C$ and (b) $h\, o.inv \leq C$ implies $h\, p.com = true$.

Corollary 1. If h is disciplined, $o \succ_C^h p$, and $h\, p.inv > type\, p$ then $h\, o.inv > C$.

Partitioning the Heap. We partition the objects in the heap in order to formalize the encapsulation boundary depicted in Sect. 2.2. Given an object $o \in dom\, h$ and class name

A with $type\,o \leq A$ we can partition h into pre-heaps Ah (the A-object), Rh (the representation of o for class A), Sh (objects owned by o at a superclass), and Fh (free from o) determined by the following conditions: Ah is the singleton $[o \mapsto h\,o]$, Rh is h restricted to the set of p with $o \succ^h_A p$, Sh is h restricted to the set of p with $o \succ^h_C p$ for some $C > A$, and Fh is the rest of h. Note that if $o \succ^h_B p$ for some proper subclass $B < A$ then $p \in dom\,Fh$. A pre-heap of the form $Ah * Rh * Sh$ is called an *island*. In these terms, dependency of admissible invariants is described in the following Proposition. As an illustration, here is the island for the left side of the situation depicted in Sect. 2.2:



Proposition 1 (island). Suppose $\mathscr{I}^C$ is an admissible invariant for C and $o \in dom\,h$ with $type\,o \leq C$. If $h = Fh * Ah * Rh * Sh$ is the partition defined above then $Fh_0 * Ah * Rh * Sh \models \mathscr{I}^C(o)$ iff $h \models \mathscr{I}^C(o)$, for all Fh_0 such that $Fh_0 * Ah * Rh * Sh$ is a heap.

The Discipline. To impose the stipulated preconditions of Table 1 we consider programs with the requisite syntactic structure (similar to formal proof outlines).

Definition 4 (properly annotated). The *annotated commands* are the subset of the category of commands where each **pack**, **unpack**, **setown**, and field update is immediately preceded by an **assert**. A *properly annotated command* is an annotated command such that each of these assertions is (or implies) the precondition stipulated in Table 1. A *properly annotated class table* is one such that each method body is properly annotated.

For any class table and family of invariants there exists a proper annotation: just add **assert** commands with the stipulated preconditions. For practical interest, of course, one wants assertions that can collectively be proved correct. The abstraction theorem depends on proper annotation but does not depend on the invariants themselves; one may take $\mathscr{I}^C = \textbf{true}$ for all C. What matters is ownership structure and the use of *inv*. We use the following [6, 21, 28].

Proposition 2. If method environment μ is disciplined then any properly annotated command S maintains $\mathscr{J}$ in the sense that for all (h,s), if $h \models \mathscr{J}$ and $(h_0, s_0) = [\![\Gamma \vdash S]\!]\mu(h,s)$ then $h_0 \models \mathscr{J}$. If CT is a properly annotated class table then the method environment $[\![CT]\!]$ is disciplined.

5 The Abstraction Theorem

5.1 Comparing Class Tables

We compare two implementations of a designated class *Abs*, in the context of a fixed but arbitrary collection of other classes, such that both implementations give rise to a well formed class table. The two versions can have completely different declarations, so long as methods of the same signatures are present — declared or inherited — in both. To simplify the additional precondition needed for reading fields, we consider programs desugared into a form like that used in Separation Logic.

Definition 5 (properly annotated for *Abs*). The *annotated commands for Abs* are those of Def. 4 with the additional restriction that no expression of the form $e.f$ occurs except in commands of the form **assert** $\mathscr{P}; x{:=}e.f$ (in particular, no field access appears in this e). The *properly annotated commands for Abs* are those that are properly annotated according to Def. 4 and moreover

- fields of *Abs* have private visibility (i.e., if $f \in dfields\,Abs$ then accesses and updates of f only occur in code of class *Abs*)
- If $\Gamma\,\mathsf{self} \neq Abs$ then field access $\Gamma \vdash x{:=}e.f$ is subject to stipulated precondition $(\forall o \mid o \succ_{Abs} e \Rightarrow o \succ_{Abs} \mathsf{self})$
- if $\Gamma\,\mathsf{self} \neq Abs$ then $\Gamma \vdash$ **pack** e **as** *Abs* is not allowed
- if $\Gamma\,\mathsf{self} \neq Abs$ then $\Gamma \vdash$ **setown** e_1 **to** (e_2, C) is subject to an additional precondition: $(\exists o \mid o \succ_{Abs} e_1) \Rightarrow C = Abs \vee (\exists o \mid o \succ_{Abs} e_2)$

The effect of the last precondition is that if e_1 is initially owned at *Abs* then after a transfer (that occurs in code outside class *Abs*) it is still owned at *Abs*.

In order to work with heap partitions, along the lines of Prop. 1, it is convenient to have notation to extract the one object in a singleton heap. We define *pickdom* by $pickdom(h) = o$ where $dom\,h = \{o\}$; it is undefined if $dom\,h$ is not a singleton.

Prop. 1 considers a single object together with its owned representation; now we consider all objects of a given class.

Definition 6 (*A*-decomposition). For any class A and heap h, the *A-decomposition* of h is the set $Fh, Ah_1, Rh_1, Sh_1 \ldots, Ah_k, Rh_k, Sh_k$ (for some $k \geq 0$) of pre-heaps, all subsets of h, determined by the following conditions:

- each $dom\,Ah_i$ contains exactly one object o and $type\,o \leq A$
- every $o \in dom\,h$ with $type\,o \leq A$ occurs in $dom\,Ah_i$ for some i;
- $dom\,Rh_i = \{p \mid o \succ^h_A p\}$ where $pickdom\,Ah_i = o$;
- $dom\,Sh_i = \{p \mid o \succ^h_{(super\,A)\uparrow} p\}$ with $pickdom\,Ah_i = o$;
- $dom\,Fh = dom\,h - (\cup i \mid dom(Ah_i * Rh_i * Sh_i))$

We say that *no A-object owns an A-object in h* provided for every o, p in $dom\,h$ if $type\,o \leq A$ and $o \succ^h_{(type\,o)\uparrow} p$ then $type\,p \not\leq A$. Def. 8 in the sequel imposes a syntactic restriction to maintain this property as an invariant, where A is the class for which two representations are compared. A consequence is that there is a unique decomposition of the heap into separate islands of the form $Ah * Rh * Sh$. We use the term "partition" even though some blocks can be empty.

Lemma 2 (*A*-partition). Suppose no A-object owns an A-object in h. Then the A-decomposition is a partition of h, that is, $h = Fh * Ah_1 * Rh_1 * Sh_1 * \ldots * Ah_k * Rh_k * Sh_k$.

To maintain the invariant that no *Abs*-object owns an *Abs*-object, we formulate a mild syntactic restriction expressed using a static approximation of ownership.

Definition 7 (may own, $\succ^{\exists}$). Given well formed CT, define $\succ^{\exists}$ to be the least transitively closed relation such that

- $D_2 \succ^{\exists} D_1$ for every occurrence of **setown** e_1 **to** (e_2, D) in a method of CT, with static types $e_1 : D_1$ and $e_2 : D_2$
- if $C \succ^{\exists} D$, $C' \leq C$ and $D' \leq D$ then $C' \succ^{\exists} D'$

If $Abs \not\succ^{\exists} Abs$ then it is a program invariant that no Abs-object owns an Abs-object (recall the definition preceding Lemma 2). This is a direct consequence of the following.

Lemma 3. It is a program invariant that if $o \succ^{h}_{C} p$ then $type\, o \succ^{\exists} type\, p$.

Definition 8 (comparable class tables). Well formed class tables CT and CT' are *comparable* with respect to class name Abs ($\neq$ Object) provided the following hold.

- $CT(C) = CT'(C)$ for all $C \neq Abs$.
- $CT(Abs)$ and $CT'(Abs)$ declare the same methods with the same signatures and the same direct superclass.
- For every method m declared in $CT(Abs)$, m is declared in $CT'(Abs)$ and has the same signature; *mutatis mutandis* for m declared in CT'.
- CT and CT' are properly annotated for Abs.
- $Abs \not\succ^{\exists} Abs$ in both CT and CT'

The last condition ensures that the Abs-decomposition of any disciplined heap is a partition, by Lemmas 2 and 3. We write $\vdash, \vdash'$ for the typing relation determined by CT, CT' respectively; similarly we write $[\![-]\!], [\![-]\!]'$ for the respective semantics.

5.2 Coupling Relations

The definitions are organized as follows. A *local coupling* is a suitable relation on islands. This induces a family of *coupling relations*, $\mathscr{R}\,\beta\,\theta$ for each category name θ and typed bijection β. Each relation $\mathscr{R}\,\beta\,\theta$ is from $[\![\theta]\!]$ to $[\![\theta]\!]'$. Here β is a bijection on locations, used to connect a heap in $[\![\mathsf{heap}]\!]$ to one in $[\![\mathsf{heap}]\!]'$. The idea is that β relates all objects except those in the Rh_i or Rh'_i blocks that have never been exposed. Finally, a *simulation* is a coupling that is preserved by all methods of Abs and holds initially.

Definition 9. A *typed bijection* is a bijective relation, β, from Loc to Loc, such that $\beta\, o\, o'$ implies $type\, o = type\, o'$ for all o, o'. A *total bijection* on h, h' is a typed bijection with $dom\, h = dom\, \beta$ and $dom\, h' = rng\, \beta$. Finally, β *fully partitions* h, h' *for Abs* if, for all $o \in dom\, h$ (resp. $o \in dom\, h'$) with $type\, o \leq Abs$, o is in $dom\, \beta$ (resp. $rng\, \beta$).

Lemma 4 (typed bijection and *Abs*-partition). Suppose β is a typed bijection with $\beta \subseteq dom\, h \times dom\, h'$ and β fully partitions h, h' for Abs. If h, h' are disciplined and partition as $h = Fh * \ldots Ah_j * Rh_j * Sh_j$ and $h' = Fh' * \ldots Ah'_k * Rh'_k * Sh'_k$ then $j = k$.

Definition 10 (equivalence for *Abs* modulo bijection). For any β we define a relation $\sim_{\beta}$ for data values, object states, heaps, and stores, in Table 6.

Table 6. Value equivalence for the designated class *Abs*. The relation for heap is the same as for pre-heap. For object states, $\sim$ is independent from the declared fields of $CT(Abs)$ and $CT'(Abs)$

$o \sim_\beta o'$	in $[\![C]\!]$	$\Leftrightarrow \beta\, o\, o' \vee o = nil = o'$
$v \sim_\beta v'$	in $[\![T]\!]$	$\Leftrightarrow v = v'$ for primitive types T
$s \sim_\beta s'$	in $[\![\mathsf{state}\,C]\!]$	$\Leftrightarrow \forall (f:T) \in \mathit{xfields}\,C \mid s f \sim_\beta s' f \vee (f:T) \in \mathit{dfields}\,\mathit{Abs}$
$s \sim_\beta s'$	in $[\![\Gamma]\!]$	$\Leftrightarrow \forall x \in \mathit{dom}\,\Gamma \mid s x \sim_\beta s' x$
$h \sim_\beta h'$	in $[\![\mathsf{pre\text{-}heap}]\!]$	$\Leftrightarrow \forall o \in \mathit{dom}\,h, o' \in \mathit{dom}\,h' \mid \beta\, o\, o' \Rightarrow h\, o \sim_\beta h' o'$
$(h,s) \sim_\beta (h',s')$	in $[\![\mathsf{heap} \otimes \Gamma]\!]$	$\Leftrightarrow h \sim_\beta h' \wedge s \sim_\beta s'$
$v \sim_\beta v'$	in $[\![\theta_\bot]\!]$	$\Leftrightarrow v = \bot = v' \vee (v \neq \bot \neq v' \wedge v \sim_\beta v'$ in $[\![\theta]\!])$
$(o,C) \sim_\beta (o',C')$	in $[\![\mathsf{owntyp}]\!]$	$\Leftrightarrow (o = nil = o') \vee (\beta\, o\, o' \wedge C = C')$
$B \sim_\beta B'$	in $[\![\mathsf{invtyp}\,C]\!]$	$\Leftrightarrow B = B'$

Equivalence hides the private fields of *Abs*. In the identity extension lemma, it is used in conjunction with the following which hides objects owned at *Abs*.

Definition 11 (encap). Suppose no A-object owns an A-object in h. Define $\mathit{encap}\,A\,h$ to be the pre-heap $Fh * Ah_1 * Sh_1 * \ldots * Ah_k * Sh_k$ where the A-partition of h is as in Lemma 2.

The most important definition is of local coupling, which is analogous to an object invariant but is a relation on pairs of pre-heaps. In Def. 2, we take an invariant $\mathscr{I}^C$ to be a predicate (set of states) and the program invariant $\mathscr{J}$ is based on the conjunction of these predicates for all objects and types —subject to *inv*, see Def. 3). By contrast, we define a local coupling $\mathscr{L}$ in terms of pre-heaps. And we are concerned with a single class, *Abs*, rather than all C. We impose the same dependency condition as in Def. 2, but in terms of pre-heaps of the form $h = Ah * Rh * Sh$. (Recall Proposition 1.)

Definition 12 (local coupling, $\mathscr{L}$). Given comparable class tables, a *local coupling* is a function, $\mathscr{L}$, that assigns to each typed bijection β a binary relation $\mathscr{L}\,\beta$ on pre-heaps that satisfies the following. First, $\mathscr{L}\,\beta$ does not depend on *inv* or *com*. Second, $\beta \subseteq \beta_0$ implies $\mathscr{L}\,\beta \subseteq \mathscr{L}\,\beta_0$. Third, for any β, h, h', if $\mathscr{L}\,\beta\, h\, h'$ then there are locations o, o' with $\beta\, o\, o'$ and $\mathit{type}\, o \leq \mathit{Abs}$ such that the *Abs* partitions of h, h' are $h = Ah * Rh * Sh$ and $h' = Ah' * Rh' * Sh'$ with

- $\mathit{pickdom}\,Ah = o$ and $\mathit{pickdom}\,Ah' = o'$
- $o \succ^{h}_{Abs} p$ for all $p \in \mathit{dom}(Rh)$ and $o' \succ^{h'}_{Abs} p'$ for all $p' \in \mathit{dom}(Rh')$
- $o \succ^{h}_{(\mathit{superAbs})\uparrow} p$ for all $p \in \mathit{dom}(Sh)$ and $o' \succ^{h'}_{(\mathit{superAbs})\uparrow} p'$ for all $p' \in \mathit{dom}(Sh')$
- If $\mathscr{L}\,\beta$ depends on f then f is in $\mathit{xfields}\,\mathit{Abs}$

The first three conditions ensure that $\mathscr{L}$ relates a single island, for an object of some subtype of *Abs*, to a single island for an object of the same type. Although $\mathscr{L}$ is unconstrained for the private fields of $CT(Abs)$ and $CT'(Abs)$, it may also depend on fields inherited from a superclass of *Abs* (but not on subclass fields). The induced coupling relation, defined below, imposes the additional constraint that fields of proper sub- and super-classes of *Abs* are linked by equivalence modulo β. Although superficially different, the notion of local coupling is closely related to admissible invariant.

Table 7. The induced coupling relation for Def. 13

$\mathscr{R}\ \beta\ \theta\ \alpha\ \alpha'$	$\Leftrightarrow \alpha \sim_\beta \alpha'$ if θ is **bool**, C, Γ, or $\mathsf{state}C$
$\mathscr{R}\ \beta\ (\mathsf{heap}\otimes\Gamma)\ (h,s)\ (h',s')$	$\Leftrightarrow \mathscr{R}\,\beta\,\mathsf{heap}\,h\,h' \wedge \mathscr{R}\,\beta\,\Gamma\ s\ s' \wedge \mathit{disciplined}(h,s) \wedge \mathit{disciplined}(h',s')$
$\mathscr{R}\ \beta\ (\mathsf{heap}\otimes T)\ (h,v)\ (h',v')$	$\Leftrightarrow \mathscr{R}\ \beta\ \mathsf{heap}\ h\ h' \wedge \mathscr{R}\ \beta\ T\ v\ v'$
$\mathscr{R}\ \beta\ (\theta_\perp)\ \alpha\ \alpha'$	$\Leftrightarrow (\alpha = \perp = \alpha') \vee (\alpha \neq \perp \neq \alpha' \wedge \mathscr{R}\ \beta\ \theta\ \alpha\ \alpha')$
$\mathscr{R}\ \beta\ (\Gamma \vdash T)\ v\ v'$	$\Leftrightarrow \forall h,s,h',s' \mid \mathscr{R}\ \beta\ (\mathsf{heap}\otimes\Gamma)\ (h,s)\ (h',s')$
	$\Rightarrow \mathscr{R}\ \beta\ T_\perp\ (v(h,s))\ (v'(h',s'))$
$\mathscr{R}\ \beta\ (C,\bar{x},\bar{T}\rightarrow T_1)\ v\ v'$	$\Leftrightarrow \forall h,s,h',s' \mid \mathscr{R}\ \beta\ (\mathsf{heap}\otimes\Gamma)\ (h,s)\ (h',s')$
	$\Rightarrow \exists \beta_0 \supseteq \beta \mid \mathscr{R}\ \beta_0\ (\mathsf{heap}\otimes T_1)_\perp\ (v(h,s))\ (v'(h',s'))$
	where $\Gamma = [\bar{x} : \bar{T}, \mathsf{self} : C]$
$\mathscr{R}\ \mathsf{menv}\ \mu\ \mu'$	$\Leftrightarrow \forall C,m,\beta \mid \mathscr{R}\ \beta\ (C,\bar{x},\bar{T}\rightarrow T)\ (\mu Cm)\ (\mu' Cm)$
	where $\mathit{mtype}(m,C) = \bar{T}\rightarrow T$ and $\mathit{pars}(m,C) = \bar{x}$

In applications, $\mathscr{L}\,\beta\,h\,h'$ would be defined as something like this: h and h' partition as islands $Ah * Rh * Sh$ and $Ah' * Rh' * Sh'$ such that $Ah * Rh * Sh \models \mathscr{I}^{Abs}$ and $Ah' * Rh' * Sh' \models \mathscr{I}'^{Abs}$ and some condition links the data structures [18]. The bijection β would not be explicit but would be induced as a property of the formula language.

A local coupling $\mathscr{L}$ induces a relation on arbitrary heaps by requiring that they partition such that islands can be put in correspondence so that pairs are related by $\mathscr{L}$.

Definition 13 (coupling relation, $\mathscr{R}$). Given local coupling $\mathscr{L}$, we define for each θ and β a relation $\mathscr{R}\,\beta\,\theta \subseteq [\![\theta]\!] \times [\![\theta]\!]'$ as follows.

For heaps h,h', we define $\mathscr{R}\ \beta\ \mathsf{heap}\,h\,h'$ iff h,h' are disciplined, $\beta \subseteq \mathit{dom}\,h \times \mathit{dom}\,h'$, and β fully partitions h,h' for Abs; moreover, if the Abs-partitions are $h = Fh * Ah_1 * Rh_1 * Sh_1 \ldots Ah_k * Rh_k * Sh_k$ and $h' = Fh' * Ah'_1 * Rh'_1 * Sh'_1 \ldots Ah'_k * Rh'_k * Sh'_k$ then (recall Lemma 4) (a) β restricts to a total bijection between $\mathit{dom}(Fh)$ and $\mathit{dom}(Fh')$; (b) $Fh \sim_\beta Fh'$; and (c) for all i,j, if $\beta\ (\mathit{pickdom}\,Ah_i)\ (\mathit{pickdom}\,Ah'_j)$ then

– β restricts to a total bijection between $\mathit{dom}(Sh_i)$ and $\mathit{dom}(Sh'_j)$
– $(Ah_i * Sh_i) \sim_\beta (Ah'_j * Sh'_j)$
– $h(\mathit{pickdom}\,Ah_i).\mathit{inv} \leq \mathit{Abs} \Rightarrow \mathscr{L}\ \beta\ (Ah_i * Rh_i * Sh_i)\ (Ah'_j * Rh'_j * Sh'_j)$

For other categories θ we define $\mathscr{R}\,\beta\,\theta$ in Table 7.

The third item under (c) is the key connection with the $\mathit{inv}/\mathit{own}$ discipline.

Under the antecedent in the definition, $(Ah_i * Sh_i) \sim_\beta (Ah'_j * Sh'_j)$ is equivalent to the conjunction of $Ah_i \sim_\beta Ah'_j$ and $Sh_i \sim_\beta Sh'_j$. And $Ah_i \sim_\beta Ah'_j$ means that the two objects o,o' agree on superclass and subclass fields (but not the declared fields of Abs); in particular, $\mathit{type}\,o = \mathit{type}\,o' \leq \mathit{Abs}$ and $Ah_i\,o.\mathit{inv} = Ah'_j\,o'.\mathit{inv}$.

The gist of the abstraction theorem is that if methods of Abs are related by $\mathscr{R}$ then all methods are. In terms of the preceding definitions, we can express quite succinctly the conclusion that all methods are related: $\mathscr{R}\ \mathsf{menv}\ [\![CT]\!]\ [\![CT']\!]'$. We want the antecedent of the theorem to be that the meaning $[\![M]\!]$ is related to $[\![M']\!]'$, for any m with declaration M in $CT(\mathit{Abs})$ and M' in $CT'(\mathit{Abs})$. Moreover, $[\![M]\!]$ depends on a method environment. Thus the antecedent of the theorem is that $[\![M]\!]\mu$ is related to $[\![M']\!]'\mu'$ for all related μ,μ'. (It suffices for μ,μ' to be in the approximation chains defining $[\![CT]\!]$ and $[\![CT']\!]'$).

5.3 Simulation and the Abstraction Theorem

Definition 14 (simulation). A simulation is a coupling $\mathscr{R}$ such that the following hold.

- ($\mathscr{L}$ is initialized) For any $C \le Abs$, and any o, o' with $\beta\, o\, o'$ and $type\, o = C$ we have $\mathscr{L}\, \beta\, h\, h'$ where $h = [o \mapsto [dom(xfields\, C) \mapsto defaults\, C]]$ and $h' = [o' \mapsto [dom(xfields'\, C) \mapsto defaults'\, C]]$.
- (methods of Abs preserve $\mathscr{R}$) For any disciplined μ, μ' such that $\mathscr{R}$ menv μ μ' we have the following for every m declared in Abs. Let $\bar{U} \to U = mtype(m, Abs)$ and $\bar{x} = pars(m, Abs)$. For every β, we have $\mathscr{R}\ \beta\ \theta\ ([\![M]\!]\mu)\ ([\![M']\!]'\mu')$ where $\theta = (Abs, \bar{x}, \bar{U} \to U)$. where M (resp. M') are as above. (We omit the similar condition for inherited methods.)

Lemma 5 (preservation by expressions). For all expressions $\Gamma \vdash e : T$ that contain no field access subexpressions, and all β, we have $\mathscr{R}\, \beta\, (\Gamma \vdash T)\ ([\![\Gamma \vdash e : T]\!])\ ([\![\Gamma \vdash e : T]\!]')$.

Lemma 6 (preservation by commands). Let μ, μ' be disciplined method environments with $\mathscr{R}$ menv μ μ' . If $\Gamma \vdash S$ is a properly annotated command for Abs, with $\Gamma\, \mathsf{self} \ne Abs$, then for all β we have the following. If $\mathscr{R}\ \beta\ (\mathsf{heap} \otimes \Gamma)\ (h, s)\ (h', s')$ and $\neg(\exists o \mid o \succ^{h}_{Abs} s(\mathsf{self}))$ and $\neg(\exists o' \mid o' \succ^{h'}_{Abs} s'(\mathsf{self}))$ then there is $\beta_0 \supseteq \beta$ such that $\mathscr{R}\ \beta_0\ (\mathsf{heap} \otimes \Gamma)_{\bot}\ (v(h, s))\ (v'(h', s'))$.

Our main result says that if methods of Abs preserve the coupling then all methods do.

Theorem 1 (abstraction).
If $\mathscr{R}$ is a simulation for comparable class tables CT, CT' then $\mathscr{R}$ menv $[\![CT]\!]$ $[\![CT']\!]'$.

6 Using the Theorem

A complete program is a command S in the context of a class table. To show equivalence between CT, S and CT', S, one proves simulation for Abs and then appeals to the abstraction theorem to conclude that $[\![S]\!]$ is related to $[\![S]\!]'$. Finally, one appeals to an *identity extension lemma* that says the relation is the identity for programs where the encapsulated representation is not visible. We choose simple formulations that can also serve to justify more specification-oriented formulations. We say that a state (h, s) is *Abs-free* if $type\, o \not\le Abs$ for all $o \in dom\, h$.

Lemma 7 (identity extension). If $\mathscr{R}\ \beta\ (\mathsf{heap} \otimes \Gamma)\ (h, s)\ (h', s')$
then $encapAbs\, (h, s) \sim_{\beta} encapAbs\, (h', s')$.

Lemma 8 (inverse identity extension). Suppose (h, s) and (h', s') are Abs-free. If $(h, s) \sim_{\beta} (h', s')$ and β is total on h, h' then $\mathscr{R}\ \beta\ (\mathsf{heap} \otimes \Gamma)\ (h, s)\ (h', s')$.

Definition 15 (program equivalence). Suppose programs $CT,(\Gamma \vdash S)$ and $CT',(\Gamma \vdash' S')$ are such that CT,CT' are comparable and properly annotated, and moreover S,S' are properly annotated. The programs are *equivalent* iff for all disciplined, *Abs*-free (h,s) and (h',s') in $[\![\mathsf{heap} \otimes \Gamma]\!]$ and all β with β total on h,h' and $(h,s) \sim_\beta (h',s')$, there is some $\beta_0 \supseteq \beta$ with $\mathit{encapAbs}([\![\Gamma \vdash S]\!]\mu(h,s)) \sim_{\beta_0} \mathit{encapAbs}([\![\Gamma \vdash' S']\!]'\mu'(h',s'))$ where $\mu = [\![CT]\!]$ and $\mu' = [\![CT']\!]'$.

Proposition 3 (simulation and equivalence). Suppose programs $CT,(\Gamma \vdash S)$ and $CT',(\Gamma \vdash' S)$ are properly annotated and $\mathscr{R}$ is a simulation from CT to CT'. If $\Gamma\,\mathsf{self} \neq$ *Abs* then the programs are equivalent.

7 Discussion

Adaptations of the inv/own Discipline. As compared with previous work on the discipline, we have imposed some additional restrictions to achieve sufficient information hiding to justify a modular rule for equivalence of class implementations. We argue that the restrictions are not onerous for practical application, though further practical experience is needed with the discipline and with our rule.

The first restriction is on field reads. Code in a client class cannot be allowed to read a field of an encapsulated representation object, although the discipline allows the existence of the reference; otherwise the client code could be representation dependent. On the other hand, a class such as *Hashtable* might be used both by clients and in the internal representation of the class *Abs* under revision; certainly the code of *Hashtable* needs to read its own fields. A distinction can be made on the basis of whether the current target object, i.e., self, is owned by an instance o of *Abs*. If it is, then we do not need the method invocation to preserve the coupling and we can allow reading of objects owned by o. If the target object is not owned by an instance of *Abs* then it should have no need to access objects owned by *Abs*. This distinction appears in the statement of Lemma 6 and it is used to stipulate a precondition for field access (see Def. 5).[4]

Because the coupling relation imposes the user-defined local coupling only when an *Abs*-object is packed, it appears necessary to restrict **pack** e **as** *Abs* to occur only in code of *Abs* in order for simulation to be checked only for that code. In the majority of known examples, packing to a class C is only done in code of C, and this is required in Leino and Müller's extension of the discipline to handle static fields.

Similar considerations apply to **setown** o **to** (p,C): care must be taken to prevent arbitrary code from moving objects across the encapsulation boundary for *Abs* in ways that do not admit modular reasoning. One would expect that code outside *Abs* cannot move objects across the *Abs*-boundary at all, but it turns out that the only problematic case is transfer out from an *Abs* island. In the unusual case that **setown** o **to** (p,C) occurs in code outside *Abs* but o is initially inside the island for some *Abs*-object, then

[4] This is unattractive in that the other stipulated preconditions mention only direct ownership whereas this one uses transitive ownership. But in practical examples, code outside *Abs* rarely has references to encapsulated objects. We believe such references can be adequately restricted using visibility control and/or lightweight confinement analyses, e.g., [31, 2].

o must end up in the island for some *Abs*-object. Our stipulated precondition says just this. In practice it seems that the obligation can be discharged by simple syntactic considerations of visibility and/or lightweight alias control.

The last restriction is that an *Abs* object cannot own other *Abs* objects. This does not preclude containers holding containers, because a container does not own its content (e.g., AQueue owns the Qnodes but not the tasks). It does preclude certain recursive situations. For example, we could allow Qnode instances to own their successors but then we could not instantiate the theory with *Abs*:=Qnode. This does not seem too important since it is Queue that is appropriate to view as an abstraction coupled by a simulation. The restriction is not needed for soundness of simulation. But absent the restriction, nested islands would require a healthiness condition on couplings (similar to the healthiness condition used by Cavalcanti and Naumann [13–Def. 5]); e.g., coupling for an instance of Qnode would need to recursively impose the same predicate on the nxt node. We disallow nested islands in the present work for simplicity and to highlight connections with separation logic.

Future Work. The discipline may seem somewhat onerous in that it uses verification conditions rather than lighter weight static analysis for control of the use of aliases. (We have to say "use of", because whereas confinement disallows certain aliases, the invariant discipline merely prevents faulty exploitation of aliases.) The Spec# project [7] is exploring the inference of annotations. For many situations, simple confinement rules and other checks are sufficient to discharge the proof obligations and this needs to be investigated for the additional obligations we have introduced. The advantage of a verification discipline over types is that, while simple cases can be checked automatically, complicated cases can be checked with additional annotations rather than simply rejected.

The generalization to a small group of related classes is important, as revisions often involve several related classes. One sort of example would be a revision of our Queue example that involves revising Qnode as well. If nodes are used only by Queue then this is subsumed by our theory, as we can consider a renamed version of Qnode that coexists with it. The more interesting situations arise in refactoring and in design patterns with tightly related configurations of multiple objects. The friend and peer dependencies of [21, 9, 28], and the flexible ownership system of Aldrich and Chambers [1] could be the basis for a generalization of our results.

References

1. J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *ECOOP*, 2004.
2. A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM*, 2002. Accepted, revision pending. Extended version of [3].
3. A. Banerjee and D. A. Naumann. Representation independence, confinement and access control. In *POPL*, 2002.
4. A. Banerjee and D. A. Naumann. Ownership transfer and abstraction. Technical Report TR 2004-1, Computing and Information Sciences, Kansas State University, 2003.

5. A. Banerjee and D. A. Naumann. State based encapsulation and generics. Technical Report CS Report 2004-11, Stevens Institute of Technology, 2004.
6. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3, 2004.
7. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS post-proceedings*, 2004.
8. M. Barnett, D. A. Naumann, W. Schulte, and Qi Sun. 99.44% pure: useful abstractions in specifications. In *ECOOP workshop on Formal Techniques for Java-like Programs*, 2004.
9. M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *Mathematics of Program Construction*, 2004.
10. P. H. M. Borba, A. C. A. Sampaio, and M. L. Cornélio. A refinement algebra for object-oriented programming. In *ECOOP*, 2003.
11. C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *POPL*, 2003.
12. J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP*, 2001.
13. A. L. C. Cavalcanti and D. A. Naumann. Forward simulation for data refinement of classes. In *Formal Methods Europe*, 2002.
14. D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, 2002.
15. D. G. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, 1998.
16. D. L. Detlefs, K. R. M. Leino, and G. Nelson. Wrestling with rep exposure. Research 156, DEC Systems Research Center, 1998.
17. J. V. Guttag and J. J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.
18. C. A. R. Hoare. Proofs of correctness of data representations. *Acta Inf.*, 1, 1972.
19. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Prog. Lang. Syst.*, 23, 2001.
20. B. Jacobs and E. Poll. Java program verification at Nijmegen: Developments and perspective. In *International Symposium on Software Security*, 2003.
21. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, 2004.
22. B. Meyer. *Object-oriented Software Construction*. Second edition, 1997.
23. I. Mijajlovic, N. Torp-Smith, and P. O'Hearn. Refinement and separation contexts. In *Foundations of Software Technology and Theoretical Computer Science (FST&TCS)*, 2004.
24. J. C. Mitchell. Representation independence and data abstraction. In *POPL*, 1986.
25. P. Müller, A. Poetzsch-Heffter, and G. Leavens. Modular invariants for object structures. Technical Report 424, ETH Zürich, Oct. 2003.
26. D. A. Naumann. Observational purity and encapsulation. In *FASE*, 2005.
27. D. A. Naumann. Verifying a secure information flow analyzer. To appear in *TPHOLS*, 2005.
28. D. A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state (extended abstract). In *LICS*, 2004.
29. P. O'Hearn, H. Yang, and J. Reynolds. Separation and information hiding. In *POPL*, 2004.
30. F. Smith, D. Walker, and G. Morrisett. Alias types. In *ESOP*, 2000.
31. J. Vitek and B. Bokowski. Confined types in Java. *Software Practice and Experience*, 31, 2001.

Consistency Checking of Statechart Diagrams of a Class Hierarchy

Vitus S.W. Lam and Julian Padget

Department of Computer Science, University of Bath
{lsw, jap}@cs.bath.ac.uk

Abstract. One of the limitations of UML is it lacks a systematic way for verifying consistency within and between models. This paper explores the intra-model consistency problem in the context of statechart diagrams. We propose an algebraic approach for determining whether the statechart diagrams of a superclass and its subclass are consistent with respect to their behaviour. The statechart diagrams are first translated into the π-calculus and then verified automatically using the Mobility Workbench.

1 Introduction

The Principle of Substitutability [1] stipulates that an instance of the supertype can always be replaced with an instance of the subtype. In the context of the object-oriented technology, the corresponding notion for subtype relation between a supertype and its subtype is inheritance.

In the Unified Modeling Language (UML) [2], a generalization relationship specifies that a subclass inherits from a superclass. The behaviour of the objects (instances) of the subclass and superclass is normally represented using statechart diagrams, but because the statechart diagrams are drawn separately, a rigorous approach for verifying consistency between the statechart diagrams and substitutability of an object of the superclass by an object of the subclass is needed. In this paper, we use weak open bisimulation of the π-calculus [3, 4] for detecting inconsistency between statechart diagrams of a superclass and its subclass. The consistency checking of statechart diagrams is carried out automatically using the Mobility Workbench (MWB) [5, 6].

The rest of the paper is organized as follows. Section 2 describes prior work in the area. Section 3 provides an overview of the main features of UML statechart diagrams and the π-calculus. The encoding of a subset of statechart diagrams in the π-calculus is presented in Section 4. Section 5 discusses the concepts of substitutability and behavioural consistency. Section 6 examines the consistency checking of statechart diagrams of classes linked with a generalization relationship using the MWB. Conclusions are given in Section 7.

2 Related Work

The consistency checking of the statechart diagrams of classes which are linked with a generalization relationship has not been fully exploited in previous studies [7, 8, 9, 10].

In [7], Sourrouille translates statechart diagrams into an object-oriented language and explores the concepts of behaviour inheritance and substitutability. In contrast, we translate statechart diagrams into the π-calculus and determines the substitutability of statechart diagrams using the weak open bisimulation of the π-calculus. Furthermore, our approach is based on a formal method instead of a programming language. Sourrouille [7] uses a simplified form of transitions, whereas our approach supports parameterized events, guard conditions and actions directly.

In [8], two types of substitutability are defined. Linear substitutability states that every trace in a superclass is a trace of its subclasses. Branching substitutability stipulates that any behaviour in a superclass is simulated by its subclasses. Branching substitutability is finer than linear substitutability. Any superclass C_1 which is branching substitutable by a subclass C_2 is linear substitutable by the subclass C_2.

Unlike [8] which defines substitutability in terms of trace containment and simulation, we define substitutability using the weak open bisimulation of the π-calculus. Our approach has some advantages over [8] as (i) it supports parameterized events, guard conditions and hierarchical statechart diagrams; and (ii) it automates the checking of substitutability through the use of the MWB.

Stumptner and Schrefl [9] verify consistency between the associated statechart diagrams of a superclass and its subclass using the concepts of observation consistency and invocation consistency as well as a set of rules. When compared with [9], our approach supports parameterized events, guard conditions and actions, and again the consistency checking is performed automatically.

In [10], consistency between the associated statechart diagrams of a superclass and its subclass is verified using Communicating Sequential Processes (CSP). The transformation from statechart diagrams into CSP is based on meta-model rules. In contrast to our approach, the transformation is not based on UML semantics and does not support parameterized events, guard conditions, actions and concurrent composite states. The approach of [10] provides only a way to verify consistency between statechart diagrams, whereas our approach covers consistency checking, equivalence checking [11] and model checking [12, 13] of statechart diagrams.

Other related studies on consistency checking include [14, 15, 16, 17]. Engels et al. [14] propose an approach for checking consistency between (i) capsule statecharts and (ii) capsule statechart and protocol statechart. Likewise, Engels et al. in [15] determine the consistency between an old capsule statechart and a new capsule statechart of UML-RT models by verifying that the old capsule statechart is a refinement of the new capsule statechart. In [16], the preservation of evolution consistency of the UML-RT models is checked using a rule-based transformation approach. The work of [17] describes a formal semantics which addresses the concurrent execution of multiple operations on an object. Practical applications of the formal semantics include checking properties of models, showing intra-model consistency and refactoring of models. Our approach, unlike [14, 15, 16, 17] which focus on UML-RT models and various types of UML diagrams, emphasizes UML statechart diagrams and consistency between UML statechart diagrams of a class hierarchy.

3 Basic Concepts

In this section, we introduce the notions and notations used throughout the paper.

3.1 Statechart Diagrams

A statechart diagram of UML depicts how an object of a class responds to various events throughout its lifetime. The two basic entities of a statechart diagram are state and transition.

Figure 1 shows the statechart diagram of a class C_2 which consists of an initial pseudostate and four basic states S_1, S_2, S_3 and S_4. The initial pseudostate is represented as a small filled circle with an outgoing transition to the default state S_1. Upon receipt of an event E_2, the transition which connects states S_1 and S_2 is fired, state S_1 is exited and state S_2 is entered. Unlike event E_2, event $E_1(p_1, p_2)$ is a parameterized event with parameters p_1 and p_2. Besides an event, a transition is optionally labelled with a guard-condition and an action. The transition which connects states S_3 and S_4 is fired when the event E_4 occurs and the guard-condition $cond_1$ holds. The action is executed and an event E_5 is sent to an object o.

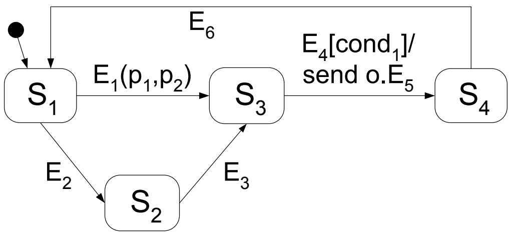


Fig. 1. Statechart diagram of class C_2

In Figure 2, state S_2 of class C_1 is a non-concurrent composite state in which only one of its substate V_1 or V_2 is active at any point of execution.

The class diagram of classes C_1 and C_2 is shown in Figure 3. The class C_1 inherits from the class C_2 and extends the class C_2 by four new methods E_7, E_8, E_9 and E_{10}.

3.2 The π-Calculus

The π-calculus is a process algebra for specifying concurrent systems in which the processes communicate over channels. As many variants of the π-calculus have

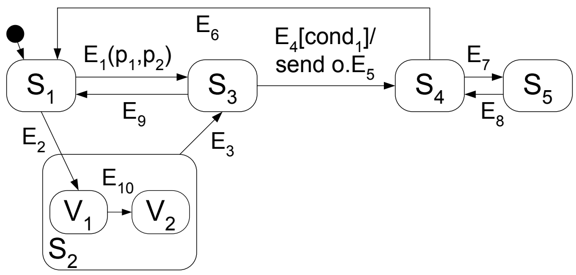


Fig. 2. Statechart diagram of class C_1

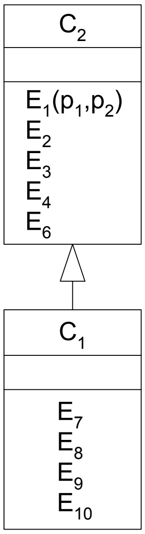


Fig. 3. Class hierarchy

been proposed, we briefly review the syntax and semantics of the π-calculus in this subsection. The reader is referred to [18, 19] for details.

We let $\mathcal{A}$ be a set of processes ranged over by P, Q, R, $\mathcal{N}$ be a set of channels (names) ranged over by x, y and $\Im$ be a set of process identifiers. The syntax and semantics of π-calculus process expressions are defined as follows:

$x(\vec{y}).P$: is an input prefix which receives channels along channel x and continues as process P with $y_1, y_2, \ldots, y_n$ replaced by the received channels. The input prefix $x().P$ is abbreviated as $x.P$.

$\overline{x}\langle\vec{y}\rangle.P$: is an output prefix which sends channels $y_1, y_2, \ldots, y_n$ along channel x and continues as process P. The output prefix $\overline{x}\langle\rangle.P$ is abbreviated as $\overline{x}.P$.

$P|Q$: represents concurrent processes P and Q are executing in parallel.

$P + Q$: represents a non-deterministic choice which either process P or Q proceeds. $\Sigma_{i=1}^{n} P_i$ abbreviates $P_1 + \ldots + P_n$.

$(\boldsymbol{\nu}\vec{x})P$: is a restriction which creates new channels $x_1, x_2, \ldots, x_n$ used for communication in process P.

$[x = y]P$: is a matching construct which proceeds as process P if channels x and y are identical; otherwise, behaves like a null process.

$\tau.P$: is an unobservable prefix which performs an internal action τ and continues as process P.

$A(x_1, x_2, \ldots, x_n) \overset{\text{def}}{=} P$: denotes a process identifier A which takes n parameters and behaves like process P. Process P may contain occurrences of A.

The input prefix $x(\vec{y}).P$ and restriction operator $(\boldsymbol{\nu}\vec{x})P$ bind $\vec{y}$ and $\vec{x}$ in P, respectively. Unlike the input prefix, the channels $\vec{y}$ in output prefix $\overline{x}\langle\vec{y}\rangle.P$ are free. The bound names and free names of P are defined as $bn(P)$ and $fn(P)$. The expression $fn(P) \cup fn(Q)$ is abbreviated as $fn(P, Q)$.

In the π-calculus, the notion of open bisimulation is used for determining whether two π-calculus processes are equivalent. We have adopted open bisimulation in this paper rather than early and late bisimulations as open bisimulation is a congruence. It preserves all π-calculus operators. In addition, the name instantiation of open bisimulation has adopted a call-by-need approach which greatly reduces the number of substitutions and provides an efficient path for tool development.

Depending on the treatment of the internal actions, open bisimulation is classified into strong open bisimulation and weak open bisimulation. Weak open bisimulation is coarser as it does not differentiate between two π-calculus processes which differ from each other in sequences of internal actions.

4 Translation of Statechart Diagrams into the π-Calculus

This section first recalls a subset of translation rules and definitions proposed in [20]. The mapping of statechart diagrams to the π-calculus is limited to notational elements which are relevant to this paper. These include events, states, guard-conditions, actions, parameterized events, non-concurrent composite states and concurrent composite states. Then an application of the translation rules is illustrated using Figures 1 and 2. The formalized execution semantics in this section extends the rule-based mapping of our previous work [20] by providing a way for transforming a parameterized event representing a method invocation into a π-calculus expression.

4.1 Translation Rules

We define $\mathcal{SC}$ as a set of statechart diagrams ranged over by F, G, H, $\mathcal{ST}$ as a set of states ranged over by S,T,V,W, $\mathcal{E}$ as a set of events ranged over by E, $\mathcal{E}_p$ as a set of parameterized events ranged over by $E(p_1, \ldots, p_n)$ and $\mathcal{TR}$ as a set of transitions ranged over by t. In addition, an infinite set of natural numbers $\mathbb{N}$ is assumed.

The translation of statechart diagrams into the π-calculus is based on the official UML semantics given in [21, 22] and a set of rules which are formalized as follows:

Rule 1. *The function $\phi_{event} : \mathcal{E} \rightarrow \mathcal{N}$ maps each event in a statechart diagram to a channel in the π-calculus.*

Rule 2. *The function $\phi_{state} : \mathcal{ST} \rightarrow \Im$ returns a unique process identifier for each state. Each process identifier $S_1(event, \vec{e}, \ldots) \in \Im$ is defined as*

$$event(x).([x = e_1] \ldots + \ldots + [x = e_n] \ldots)$$

where $\vec{e}$ stands for $e_1, \ldots, e_n$ and $\forall a \in \{\vec{e}\}.\phi^{-1}_{event}(a) \in \mathcal{E}$.

Rule 1 specifies that an event is modelled as a channel in the π-calculus. The inverse of ϕ_{event} denoted by ϕ^{-1}_{event} is a function from $\mathcal{N}$ to $\mathcal{E}$. Rule 2 stipulates that a state is encoded in the π-calculus as a process. The process is regarded as an event processor of the statechart diagram which handles each dispatched event according to the UML semantics. It determines what the event is by using a number of matching constructs.

We define $\mathcal{A}_{in} = \{x(\vec{y})|x, \vec{y} \in \mathcal{N}\}$ to be a set of input actions and $\mathcal{A}_{out} = \{\overline{x}\langle\vec{y}\rangle|x, \vec{y} \in \mathcal{N}\}$ to be a set of output actions.

Definition 1. *The function* arity: $(\mathcal{A}_{in} \cup \mathcal{A}_{out}) \rightarrow \mathbb{N}$ *returns the number of channels which an input or output action takes as parameters.*

Rule 3. *A mapping between guard-conditions and output actions is defined as $\phi_{guard} : GCond \rightarrow \{\alpha|\alpha \in \mathcal{A}_{out} \wedge arity(\alpha) = 1\}$ where $GCond$ is a set of guard-conditions. The Boolean value of a guard-condition is tested by*

$$\overline{g}\langle x\rangle.x(y).([y = true] \cdots + [y = false] \cdots)$$

where $g, x, y, true, false \in \mathcal{N}$ and $\phi^{-1}_{guard}(\overline{g}\langle x\rangle) \in GCond$.

Rule 4. *Each action representing the invocation of an operation or the sending of a signal to an object is related to an output action in the π-calculus by $\phi_{action} : Act \rightarrow \mathcal{A}_{out}$ where Act is a set of actions.*

Rules 3 and 4 say that the guard-condition and action of a transition are both represented as an output action. Rule 3 defines how a guard-condition and its evaluation are formalized. The encoding uses two matching constructs to distinguish between the two truth values.

Rule 5. *The function $\phi_{pevent} : \mathcal{E}_p \rightarrow \mathcal{N}$ maps a parameterized event to a channel.*

Rule 6. *The receipt of a parameterized event $E_1(p_1, \ldots, p_n) \in \mathcal{E}_p$ is encoded as:*

$$event(x).([x = e_1]x(p_1, \ldots, p_n). \cdots + \cdots + [x = e_n] \cdots)$$

Rule 5 states that a parameterized event is translated into a channel. The parameters $p_1, \ldots, p_n$ of the parameterized event E_1 are received along the event channel e_1 as defined by Rule 6.

Definition 2. *The function* substates: $\mathcal{ST} \rightarrow 2^{\mathcal{ST}}$ *returns the direct substates that are directly contained in a composite state.*

Rule 7. *A non-concurrent composite state S_1 and its active substate V_1 are denoted as $\phi_{state}(S_1)|\phi_{state}(V_1)$ where $V_1 \in substates(S_1)$ and $\phi_{state}(S_1)$ and $\phi_{state}(V_1)$ are defined by:*

$$\begin{aligned}
&S_1(step, event_S, \vec{e}, event_V, pos, neg) \overset{\text{def}}{=} \\
&\quad event_S(x).(\boldsymbol{\nu} ack) \\
&\quad \overline{event_V}\langle x\ ack\rangle.ack(y).([y = pos]\overline{step}. \cdots + [y = neg]\overline{step}. \cdots)
\end{aligned}$$

$$\begin{aligned}
&V_1(event_V, \vec{e}, pos, neg) \overset{\text{def}}{=} \\
&\quad event_V(x\ ack). \\
&\quad ([x = e_1]\overline{ack}\langle value_1\rangle. \cdots + \cdots + \\
&\quad\ [x = e_n]\overline{ack}\langle value_n\rangle. \cdots)
\end{aligned}$$

where $value_i \in \{pos, neg\}$ for $i = 1, \ldots, n$.

Rule 8. *A concurrent composite state S_1 and its active substates $V_1, \ldots, V_n$ which are located in n different orthogonal regions are represented in the π-calculus as $\phi_{state}(S_1)|\ \phi_{state}(V_1)| \ldots |\ \phi_{state}(V_n)$ where $\bigwedge_{i=1}^{n} V_i \in substates(S_1)$ and $\phi_{state}(S_1)$, $\phi_{state}(V_1)$, $\ldots$, $\phi_{state}(V_n)$ are defined by:*

$$\begin{aligned}
&S_1(step, event_S, \vec{e}, event_{V_1}, \ldots, event_{V_n}, pos, neg, \ldots) \overset{\text{def}}{=} \\
&\quad event_S(x).(\boldsymbol{\nu}\overrightarrow{ack})\overline{event_{V_1}}\langle x\ ack_1\rangle. \cdots . \\
&\quad \overline{event_{V_n}}\langle x\ ack_n\rangle.ack_1(y_1). \cdots .ack_n(y_n). \cdots
\end{aligned}$$

$$\begin{aligned}
&V_i(event_{V_i}, \vec{e}, pos, neg, \ldots) \overset{\text{def}}{=} \\
&\quad event_{V_i}(x\ ack). \\
&\quad ([x = e_1]\overline{ack_i}\langle value_1\rangle. \cdots + \cdots + \\
&\quad\ [x = e_n]\overline{ack_i}\langle value_n\rangle. \cdots)
\end{aligned}$$

where $1 \leq i \leq n$ and $value_i \in \{pos, neg\}$ for $i = 1, \ldots, n$.

Rules 7 and 8 specify that a composite state and its active direct substates [21] are denoted as processes which are running in parallel. A non-concurrent composite state is regarded as a special case of a concurrent composite state in which there is only one orthogonal region. The composite state broadcasts any received events to its substates. As the substates process the received event before the composite state, the lowest-first firing priority of UML semantics is preserved in our translation. The end of a run-to-completion step [21, 22] is encoded as an output action $\overline{step}$ (Rule 7).

4.2 Application of the Translation Rules

An illustration of how various notational elements including a basic state, an event, a parameterized event, a guard-condition, an action, a non-concurrent composite state, a substate and an interlevel transition are represented in the π-calculus is shown here. To improve the readability of the π-calculus specifications, some abbreviations are defined as follows:

$$\begin{aligned}\widetilde{e_{C_2}} &= e_1, e_2, e_3, e_4, e_5, e_6\\ \widetilde{e_{C_1}} &= e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}\\ \widetilde{ack} &= pos, neg\end{aligned}$$

According to Rules 1, 2, 5 and 6, the basic state S_1, the event E_2 and the parameterized event $E_1(p_1, p_2)$ of class C_2 in Figure 1 are modelled in the π-calculus as:

$$\begin{aligned}&S_1^{C_2}(step, events_S, \widetilde{e_{C_2}}, cond_1, ins_0) \overset{\text{def}}{=}\\ &\quad events_S(x).\\ &\quad([x = e_1]x(p_1, p_2).\overline{step}.\\ &\qquad S_3^{C_2}(step, events_S, \widetilde{e_{C_2}}, cond_1, ins_0) +\\ &\quad [x = e_2]\overline{step}.S_2^{C_2}(step, events_S, \widetilde{e_{C_2}}, cond_1, ins_0) +\\ &\quad \Sigma_{i\in\{3,\ldots,6\}}[x = e_i]\overline{step}.S_1^{C_2}(step, events_S, \widetilde{e_{C_2}}, cond_1, ins_0))\end{aligned}$$

The process $S_1^{C_2}$ receives an event along $events_S$ and determines what the received event is. If the event E_1, modelled as channel e_1, is received, it inputs two parameters p_1 and p_2 along e_1, sends a signal on channel *step* and continues as process $S_3^{C_2}$. Upon receipt of the event E_2, it sends a signal on channel *step* and evolves to the process $S_2^{C_2}$. Otherwise, it outputs a signal on channel *step* and continues as itself.

Based on Rules 1–4, the basic state S_3 and the transition labelled with an event E_4, a guard-condition $cond_1$ and an action $send\ o.E_5$ (Figure 1) are represented as:

$$\begin{aligned}&S_3^{C_2}(step, events_S, \widetilde{e_{C_2}}, cond_1, ins_0) \overset{\text{def}}{=}\\ &\quad events_S(x).\\ &\quad([x = e_4](\boldsymbol{\nu} true\ false)\overline{cond_1}\langle true\ false\rangle.\\ &\quad(true.\overline{ins_0}\langle e_5\rangle.\overline{step}.\\ &\qquad S_4^{C_2}(step, events_S, \widetilde{e_{C_2}}, cond_1, ins_0) +\\ &\quad false.\overline{step}.S_3^{C_2}(step, events_S, \widetilde{e_{C_2}}, cond_1, ins_0)) +\\ &\quad \Sigma_{i\in\{1,2,3,5,6\}}[x = e_i]\overline{step}.S_3^{C_2}(step, events_S, \widetilde{e_{C_2}}, cond_1, ins_0))\end{aligned}$$

The process $S_3^{C_2}$ waits on channel $events_S$ for the event E_4, creates a pair of channels $true$ and $false$ and outputs them on channel $cond_1$. On receiving a signal along $true$, it sends the event E_5 on channel ins_0, outputs a signal on channel *step* and proceeds as process $S_4^{C_2}$. Upon receipt of a signal along $false$, it outputs a signal on channel *step* and proceeds as itself.

Applying Rules 1, 2, 5, 6 and 7, the basic state S_1, the non-concurrent composite state S_2, the substate V_1 and the interlevel transition between S_1 and V_1 in Figure 2 are translated into a π-calculus specification defined below:

$$\begin{aligned}
&S_1^{C_1}(step, events_S, \widetilde{e_{C_1}}, cond_1, ins_0, \widetilde{ack}) \stackrel{\text{def}}{=} \\
&\quad events_S(x). \\
&\quad ([x = e_1]x(p_1, p_2).\overline{step}.S_3^{C_1}(step, events_S, \widetilde{e_{C_1}}, cond_1, ins_0, \widetilde{ack}) + \\
&\quad\ [x = e_2]\overline{step}.(\boldsymbol{\nu} event_{sub})(S_2^{C_1}(step, events_S, \widetilde{e_{C_1}}, cond_1, ins_0, event_{sub}, \widetilde{ack})| \\
&\quad\ \ V_1^{C_1}(event_{sub}, \widetilde{e_{C_1}}, cond_1, ins_0, \widetilde{ack})) + \\
&\quad\ \Sigma_{i \in \{3,\ldots,10\}}[x = e_i]\overline{step}.S_1^{C_1}(step, events_S, \widetilde{e_{C_1}}, cond_1, ins_0, \widetilde{ack}))
\end{aligned}$$

On receiving the event E_2, the process $S_1^{C_1}$ outputs a signal along *step* and continues as two concurrent processes $S_2^{C_1}$ and $V_1^{C_1}$ representing the non-concurrent composite state and the active substate.

Using a similar approach, other parts of Figures 1 and 2 are translated into their equivalent π-calculus representations. For reasons of space, we omit the details here.

5 Substitutability and Behavioural Consistency

This section first introduces the notions of extension, substitutability of objects, substitutability of statechart diagrams and behavioural consistency of statechart diagrams. Then we discuss how the behavioural consistency of statechart diagrams of a class hierarchy is checked using weak open bisimulation.

We define $\mathcal{C}$ as a set of classes ranged over by C and describe now a formal definition of class.

Definition 3 (Class). *Given a class $C \in \mathcal{C}$, the class C is a 2-tuple $C = (\Omega_A^C, \Omega_M^C)$ where (i) Ω_A^C is a set of attributes and (ii) Ω_M^C is a set of methods.*

We let Ω_A^C be a set of attributes and Ω_M^C be a set of methods of a subclass $C \in \mathcal{C}$. The sets of new attributes and new methods defined in the subclass C are denoted as Δ_A^C and Δ_M^C where $\Delta_A^C \subseteq \Omega_A^C$ and $\Delta_M^C \subseteq \Omega_M^C$.

Definition 4. *The function $\delta : \mathcal{C} \rightarrow \mathcal{SC}$ maps each class to an associated statechart diagram.*

Definition 5. *The function* States*: $\mathcal{SC} \rightarrow 2^{\mathcal{ST}}$ defined by States(F) $= \{S \mid S$ is a state of the statechart diagram $F\}$ returns the set of states of a statechart diagram $F \in \mathcal{SC}$.*

Definition 6. *The function $\gamma : \mathcal{ST} \rightarrow 2^{\Omega_A^C}$ where $S_i \in \mathcal{ST}$ for $i = 1, \ldots, n$ such that $States^{-1}(\bigcup_{i=1}^{n}\{S_i\}) = F$ and $\delta^{-1}(F) = C$ returns a set of attributes which are grouped together as a state of the statechart diagram F of a class C.*

Definition 7 (Extension). *Given $C_1, C_2 \in \mathcal{C}$ and C_1 is a subclass of C_2, the class C_1 extends the class C_2, written $C_1 \Rrightarrow_E C_2$, iff (i) $\Omega_A^{C_2} \subseteq \Omega_A^{C_1}$; (ii) $\Omega_M^{C_2} \subseteq \Omega_M^{C_1}$; (iii) $\Delta_A^{C_1} \cap \Omega_A^{C_2} = \emptyset$; and (iv) $\Delta_M^{C_1} \cap \Omega_M^{C_2} = \emptyset$.*

Corollary 1. *The relation $\Rrightarrow_E$ is transitive.*
Proof. *Let $C_1, C_2, C_3 \in \mathcal{C}$. Suppose $C_1 \Rrightarrow_E C_2$ and $C_2 \Rrightarrow_E C_3$. Since $C_1 \Rrightarrow_E C_2$ and $C_2 \Rrightarrow_E C_3$, it follows that $\Omega_M^{C_2} \subseteq \Omega_M^{C_1}$ and $\Omega_M^{C_3} \subseteq \Omega_M^{C_2}$. Since $\subseteq$ is transitive, we get $\Omega_M^{C_3} \subseteq \Omega_M^{C_1}$. Since $\Omega_M^{C_3} \subseteq \Omega_M^{C_2}$ and $\Delta_M^{C_1} \cap \Omega_M^{C_2} = \emptyset, \Delta_M^{C_1} \cap \Omega_M^{C_3} = \emptyset$. A similar argument holds for attributes. Thus, $\Rrightarrow_E$ is transitive.*

Definition 8 (Generalization). *Given $C_1, C_2 \in \mathcal{C}$ and C_1 is a subclass of C_2, the class C_1 generalizes the class C_2, written $C_1 \Rrightarrow_G C_2$, iff (i) $\Omega_A^{C_2} \subseteq \Omega_A^{C_1}$; (ii) $\Omega_M^{C_2} \subseteq \Omega_M^{C_1}$; (iii) $(\Delta_A^{C_1} \cap \Omega_A^{C_2} = \emptyset) \vee (\Delta_A^{C_1} \cap \Omega_A^{C_2} \neq \emptyset)$; and (iv) $(\Delta_M^{C_1} \cap \Omega_M^{C_2} = \emptyset) \vee (\Delta_M^{C_1} \cap \Omega_M^{C_2} \neq \emptyset)$.*

Proposition 1. *Let $C_1, C_2 \in \mathcal{C}$. If $C_1 \Rrightarrow_E C_2$ then $C_1 \Rrightarrow_G C_2$.*
Proof. *Let C_1 and C_2 be arbitrary classes. Suppose $C_1 \Rrightarrow_E C_2$. Since $C_1 \Rrightarrow_E C_2$, it follows that $\Omega_A^{C_2} \subseteq \Omega_A^{C_1}, \Omega_M^{C_2} \subseteq \Omega_M^{C_1}, \Delta_A^{C_1} \cap \Omega_A^{C_2} = \emptyset$ and $\Delta_M^{C_1} \cap \Omega_M^{C_2} = \emptyset$. Consider the clause $\Delta_A^{C_1} \cap \Omega_A^{C_2} = \emptyset$. Then clearly C_1 and C_2 also satisfy the clause $(\Delta_A^{C_1} \cap \Omega_A^{C_2} = \emptyset) \vee (\Delta_A^{C_1} \cap \Omega_A^{C_2} \neq \emptyset)$. A similar argument holds true for the clause $\Delta_M^{C_1} \cap \Omega_M^{C_2} = \emptyset$. Since $C_1 \Rrightarrow_G C_2$ satisfies the other two clauses $\Omega_A^{C_2} \subseteq \Omega_A^{C_1}$ and $\Omega_M^{C_2} \subseteq \Omega_M^{C_1}$ according to the definition of $\Rrightarrow_G$, we can conclude that if $C_1 \Rrightarrow_E C_2$ then $C_1 \Rrightarrow_G C_2$.*

Relationships that may exist between classes include extension (Definition 7) and generalization (Definition 8). Unlike an extension [23] which only adds more attributes and methods to a superclass, a generalization allows attribute and method overriding. An extension is finer than a generalization as specified in Proposition 1.

Definition 9 (Substitutability of Objects). *Given $C_1, C_2 \in \mathcal{C}$, any object o_1 of the class C_1 is substitutable for an object o_2 of the class C_2, written $o_1 \preceq_{obj} o_2$, iff $C_1 \Rrightarrow_E C_2$ or $C_1 \Rrightarrow_G C_2$ holds.*

Definition 10 (Substitutability of Statechart Diagrams). *Given $C_1, C_2 \in \mathcal{C}, C_1 \Rrightarrow_E C_2, \delta(C_1) = F_1$ and $\delta(C_2) = F_2$, the statechart diagram F_1 which is associated with the class C_1 is substitutable for the statechart diagram F_2 which is associated with the class C_2, written $F_1 \preceq_{sc} F_2$, if the extra attributes and methods declared in the class C_1 are specified as invisible.*

Substitutability [7, 8, 10, 23] of objects (Definition 9) is an important concept in object-oriented technology which ensures that an object of a subclass is compatible with an object of a superclass. Similarly, we introduce the concept substitutability of statechart diagrams (Definition 10) for specifying the compatibility between the statechart diagrams of a subclass and its superclass due to an extension. Hiding the extended attributes and methods in a subclass guarantees the substitutability of statechart diagrams. Both extension and generalization imply the substitutability of objects, whereas extension implies the substitutability of statechart diagrams.

Definition 11. (Behavioural Consistency of Statechart Diagrams) *Given $C_1, C_2 \in \mathcal{C}, C_1 \Rrightarrow_E C_2, \delta(C_1) = F_1$ and $\delta(C_2) = F_2$, the statechart diagram F_1 is behaviourally consistent with the statechart diagrams F_2, written as $F_1 \precsim_{sc} F_2$, iff $F_1 \preceq_{sc} F_2$.*

The behaviour of F_1 and F_2 is consistent as extended attributes and methods of F_1 are hidden according to the definition of the substitutability of statechart diagrams.

Corollary 2. *The relation $\preceq_{sc}$ is transitive.*
Proof. Let $\delta(C_1) = F_1, \delta(C_2) = F_2$ and $\delta(C_3) = F_3$. Suppose $F_1 \preceq_{sc} F_2$ and $F_2 \preceq_{sc} F_3$. Since $F_1 \preceq_{sc} F_2$ and $F_2 \preceq_{sc} F_3$, it follows that $C_1 \Rrightarrow_E C_2$ and $C_2 \Rrightarrow_E C_3$. Since $\Rrightarrow_E$ is transitive, we get $C_1 \Rrightarrow_E C_3$ and we can conclude that $F_1 \preceq_{sc} F_3$. Thus, $\preceq_{sc}$ is transitive.

Corollary 3. *The relation $\precsim_{sc}$ is transitive.*
Proof. By Definition 11 and Corollary 2.

The substitutability and behavioural consistency of statechart diagrams are transitive and any three statechart diagrams related by extensions are compatible.

Proposition 2. *Consider two objects o_1 and o_2 of classes C_1 and C_2 and the respective statechart diagrams F_1 and F_2 of the two classes. If $F_1 \preceq_{sc} F_2$ then $o_1 \preceq_{obj} o_2$.*

Proof. Suppose $F_1 \preceq_{sc} F_2$. Since $F_1 \preceq_{sc} F_2$, it follows that $C_1 \Rrightarrow_E C_2$. Therefore, $o_1 \preceq_{obj} o_2$. Thus, if $F_1 \preceq_{sc} F_2$ then $o_1 \preceq_{obj} o_2$.

Proposition 3. *Consider two objects o_1 and o_2 of classes C_1 and C_2 and the respective statechart diagrams F_1 and F_2 of two classes. If $F_1 \precsim_{sc} F_2$ then $o_1 \preceq_{obj} o_2$.*

Proof. Follows directly from Definition 11 and Proposition 2.

Proposition 2 says that an object o_1 is substitutable for an object o_2 whenever the associated statechart diagram F_1 of the object o_1 is substitutable for the associated statechart diagram F_2 of the object o_2. Similarly, the behavioural consistency of statechart diagrams implies the substitutability of their corresponding objects as stated in Proposition 3.

Next, we define the name substitution function [3, 4, 19] and a number of redexes [5, 19]. Then we recall the notion of weak open bisimulation [6, 19, 24, 25] in the π-calculus.

Definition 12. *The name substitution function* $\sigma : \mathcal{N} \to \mathcal{N}$, *written* $\{\vec{x}/\vec{y}\}$, *replaces each* $y_i \in \mathcal{N}$ *by* $x_i \in \mathcal{N}$ *for* $1, \ldots, n$.

The syntax and semantics of redexes used in the definition of weak open bisimulation are given below:

$$\begin{array}{ll} P \xrightarrow{\alpha} P' & : \text{the execution of action } \alpha \text{ and process } P \text{ becomes } P'. \\ P \Longrightarrow P' & : \text{process } P \text{ becomes } P' \text{ after zero or more internal actions.} \\ P \overset{\alpha}{\Longrightarrow} P' & : \text{is equivalent to } P \Longrightarrow\xrightarrow{\alpha}\Longrightarrow P'. \\ P \overset{\hat{\alpha}}{\Longrightarrow} P' & : \begin{cases} P \overset{\alpha}{\Longrightarrow} P' \ if \ \alpha \neq \tau \\ \\ P \Longrightarrow P' \ if \ \alpha = \tau \end{cases} \end{array}$$

Definition 13 (Weak Open Bisimulation [19]). *A symmetric binary relation* $\mathcal{R}$ *on processes is a weak open bisimulation if* $(P, Q) \in \mathcal{R}$ *implies* $\forall\sigma$ *whenever* $P\sigma \xrightarrow{\alpha} P'$ *where* $bn(\alpha) \cap fn(P\sigma, Q\sigma) = \emptyset$ *then,* $\exists Q' : Q\sigma \overset{\hat{\alpha}}{\Longrightarrow} Q'$ $\wedge(P', Q') \in \mathcal{R}$. *P is weakly open bisimilar to Q, written* $P \dot{\approx}_o Q$, *if they are related by a weak open bisimulation.*

Definition 14. *The translation of statechart diagrams into the* π*-calculus is defined by the function* $\phi : \mathcal{SC} \to 2^{\Im}$ *which represents a group of translation functions as specified by Rules 1–8.*

Proposition 4. *Let* $\delta(C_1) = F_1, \delta(C_2) = F_2, \mathcal{MC}$ *be a set of matching constructs and* $\mathbb{Z}^+$ *be a set of positive integers. If* $C_1 \Rrightarrow_E C_2$ *and* $\phi(F_2) \dot{\approx}_o (\boldsymbol{\nu}\phi(m_1)\ \phi(m_2)\ \ldots\ \phi(m_n))\ \phi(F_1)$ *then* $F_1 \precsim_{sc} F_2$ *where* $\bigcup_{i=1}^{n}\{m_i\} = \Delta_M^{C_1}$ *and the execution of the* π*-calculus expression* $events_S(x).[x = \phi(m_i)]\overline{step}$ *becomes* $\phi(S_i)$ *for* $i = 1, \ldots, n$ *such that (i)* $events_S(x) \in \mathcal{A}_{in}$, *(ii)* $[x = \phi(m_i)] \in \mathcal{MC}$, *(iii)* $\overline{step} \in \mathcal{A}_{out}$, *(iv)* $S_i \in \mathcal{ST}$, *(v)* $\gamma(S_i) = \{a_j | a_j \in \Delta_A^{C_1} \wedge j \in \mathbb{Z}^+\}$ *and (vi)* $\bigcup_{i=1}^{n}\{S_i\} \subseteq States(\delta(C_1))$.
Proof. By Definitions 10 and 11.

Proposition 4 stipulates that to determine the behavioural consistency of statechart diagrams, we translate the statechart diagrams into π-calculus specifications and hide the extended methods explicitly and extended attributes implicitly using a restriction. Then we test whether the two π-calculus specifications are weakly open bisimilar.

6 Consistency Checking Using the MWB

The MWB is an automated software tool for the π-calculus. It provides an environment for analyzing concurrent systems which have dynamically evolving communication topologies. Equivalence-checking commands for determining whether two processes specified in the π-calculus are related by various weak open bisimulations are supported.

The syntactical differences between the MWB and the π-calculus are minor. The restriction operator $\boldsymbol{\nu}$ and output action $\overline{x}$ are represented in the MWB as ˆ and 'x, respectively. In the MWB, a process is defined in the same way as in the π-calculus except that it is preceded by the keyword *agent*.

To check whether the statechart diagram of class C_1 is behaviourally consistent with the statechart diagram of class C_2 (Figures 1 and 2), the channels e_7, e_8, e_9 and e_{10} representing extended methods declared in class C_1 are hidden using a restriction as follows:

$$\begin{aligned}&(\boldsymbol{\nu} e_7\ e_8\ e_9\ e_{10}\ \widetilde{ack})\\&S_1^{C_1}(step, event_S, \widetilde{e_{C_1}}, cond_1, ins_0, \widetilde{ack})\end{aligned}$$

We then proceed to verify whether $S_1^{C_2}(step, event_S, \widetilde{e_{C_2}}, cond_1, ins_0)$ as discussed in Section 4 is weakly open bisimilar to $(\boldsymbol{\nu} e_7\ e_8\ e_9\ e_{10}\ \widetilde{ack})S_1^{C_1}(step, event_S, \widetilde{e_{C_1}}, cond_1, ins_0, \widetilde{ack})$ i.e. whether the equivalence

$$\begin{aligned}&S_1^{C_2}(step, event_S, \widetilde{e_{C_2}}, cond_1, ins_0) \dot{\approx}_o\\&(\boldsymbol{\nu} e_7\ e_8\ e_9\ e_{10}\ \widetilde{ack})\\&S_1^{C_1}(step, event_S, \widetilde{e_{C_1}}, cond_1, ins_0, \widetilde{ack})\end{aligned}$$

holds using the MWB.

As shown in Figure 4, the MWB command *input* imports the π-calculus specifications of the statechart diagrams of classes C_1 and C_2 from the file *statecharts_of_C1_C2.pi* under the directory *inheritance*. The processes

$$\begin{aligned}&S\_1\_C2(step, event\_S, e\_1, e\_2, e\_3, e\_4, e\_5, e\_6,\\&\quad cond\_1, ins\_o)\end{aligned}$$

and

$$\begin{aligned}&(\hat{\ } e\_7)(\hat{\ } e\_8)(\hat{\ } e\_9)(\hat{\ } e\_10)(\hat{\ } pos)(\hat{\ } neg)\\&S\_1\_C1(step, event\_S, e\_1, e\_2, e\_3, e\_4, e\_5, e\_6,\\&\quad e\_7, e\_8, e\_9, e\_10, cond\_1, ins\_o, pos, neg)\end{aligned}$$

representing

$$S_1^{C_2}(step, event_S, \widetilde{e_{C_2}}, cond_1, ins_0)$$

and

$$(\boldsymbol{\nu} e_7\ e_8\ e_9\ e_{10}\ \widetilde{ack})$$
$$S_1^{C_1}(step, event_S, \widetilde{e_{C_1}}, cond_1, ins_0, \widetilde{ack})$$

are then checked whether they are weakly open bisimilar given that channels $step, event_S, e_1, e_2, e_3, e_4, e_5, e_6, cond_1$ and ins_o are distinct using the MWB command *weqd*.

The two processes (agents) are related by a weak open bisimulation and the size of the weak open bisimulation is 19. The time taken for the consistency check was 0.078 seconds. The real time elapsed was measured using the MWB *time* command. The test was performed using MWB 3.122 running under Windows XP Professional operating system on a 2.4 GHz Pentium PC with 512MB of RAM.

```
 The Mobility Workbench
 (MWB'97, polyadic version 3.122, built Mon Apr 21 23:02:07 2003)

MWB>input "inheritance\statecharts_of_C1_C2.pi"
MWB>weqd \
(step,event_S,e_1,e_2,e_3,e_4,e_5,e_6,cond_1,ins_o) \
S_1_C2(step,event_S,e_1,e_2,e_3,e_4,e_5,e_6,cond_1, \
       ins_o) \
((^e_7)(^e_8)(^e_9)(^e_10)(^pos)(^neg) \
S_1_C1(step,event_S,e_1,e_2,e_3,e_4,e_5,e_6,e_7,e_8, \
       e_9,e_10,cond_1,ins_o,pos,neg))
The two agents are related.
Relation size = 19.
```

Fig. 4. Consistency checking of statechart diagrams

7 Conclusions

Ensuring the consistency of models is a non-trivial challenge for the software engineering field. This paper has examined the concepts substitutability of objects, substitutability of statechart diagrams and behavioural consistency of statechart diagrams. We have presented a new algebraic methodology for verifying whether the statechart diagrams of classes linked with a generalization relationship are consistent. The statechart diagrams are encoded in the π-calculus and the consistency of the statechart diagrams is verified using the MWB. We have plans to extend the methodology for checking the preservation of consistency between different types of UML models.

References

1. P. Wegner and S.B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In *ECOOP '88*, LNCS 322, pages 55–77, 1988.
2. OMG. OMG Unified Modeling Language specification version 1.5, March 2003. `http://www.omg.org`; accessed January 20, 2005.
3. R. Milner, J. Parrow, and D. Walker. A calculus of mobile process (Parts I and II). *Information and Computation*, 100:1–77, 1992.
4. R. Milner. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification, Proceedings of International NATO Summer School*, volume 94, pages 203–246. Springer-Verlag, 1993.
5. B. Victor and F. Moller. The mobility workbench: A tool for the π-calculus. In *CAV '94*, LNCS 818, pages 428–440, 1994.
6. B. Victor. *A Verification Tool for the Polyadic π-Calculus.* Department of Computer Systems, Uppsala University, 1994. Licentiate thesis.
7. J.L. Sourrouille. UML behaviour: Inheritance and implementation in current object-oriented languages. In *UML '99*, LNCS 1723, pages 457–472, 1999.
8. D. Harel and O. Kupferman. On the behavioral inheritance of state-based objects. In *TOOLS 34*, pages 83–94. IEEE Computer Society, 2000.
9. M. Stumptner and M. Schrefl. Behaviour consistent inheritance in UML. In *ER2000*, LNCS 1920, pages 527–542, 2000.
10. G. Engels, R. Heckel, and J.M. Küster. Rule-based specification of behavioural consistency based on the UML meta-model. In *UML 2001*, LNCS 2185, pages 272–286, 2001.
11. V.S.W. Lam and J. Padget. Analyzing equivalences of UML statechart diagrams by structural congruence and open bisimulations. In *Proceedings of 2003 IEEE Symposium on Human Centric Computing Languages and Environments*, pages 137–144. IEEE Computer Society, 2003.
12. V.S.W. Lam and J. Padget. Symbolic model checking of UML statechart diagrams with an integrated approach. In *Proceedings of Eleventh IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 337–346. IEEE Computer Society, 2004.
13. V.S.W. Lam and J. Padget. Formal specification and verification of the SET/A protocol with an integrated approach. In *Proceedings of 2004 IEEE International Conference on E-Commerce Technology*, pages 229–235. IEEE Computer Society, 2004.
14. G. Engels, J.M. Küster, R. Heckel, and L. Groenewegen. A methodology for specifying and analyzing consistency of object-oriented behavioral models. In *ESEC/SIGSOFT FSE*, pages 186–195. ACM Press, 2001.
15. G. Engels, J.M. Küster, R. Heckel, and L. Groenewegen. Towards consistency-preserving model evolution. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 129–132. ACM Press, 2002.
16. G. Engels, R. Heckel, J.M. Küster, and L. Groenewegen. Consistency-preserving model evolution through transformations. In *UML 2002*, LNCS 2460, pages 212–226, 2002.
17. J. Davies and C. Crichton. Concurrency and refinement in the Unified Modeling Language. *Formal Aspects of Computing*, 15(2–3):118–145, 2003.
18. R. Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.

19. J. Parrow. An introduction to the π-calculus. In A. Bergstra, J.A. Ponse and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 8, pages 479–543. Elsevier Science, 2001.
20. V.S.W. Lam and J. Padget. On execution semantics of UML statechart diagrams using the π-calculus. In *Proceedings of the International Conference on Software Engineering Research and Practice*, pages 877–882. CSREA Press, 2003.
21. OMG. OMG Unified Modeling Language specification version 1.5, March 2003. `http://www.omg.org`; accessed January 20, 2005.
22. OMG. UML 2.0 superstructure specification, August 2003. `http://www.omg.org`; accessed January 20, 2005.
23. J. Ebert and G. Engels. Structural and behavioural views on OMT-classes. In *Object-Oriented Methodologies and Systems*, LNCS 858, pages 142–157, 1994.
24. D. Sangiorgi. A theory of bisimulation for the π-calculus. In *CONCUR '93*, LNCS 715, pages 127–142, 1993.
25. P. Quaglia. The π-calculus: Notes on labelled semantics. *Bulletin of the EATCS*, 68, June 1999.

Towards Type Inference for JavaScript[⋆]

Christopher Anderson[1], Paola Giannini[2], and Sophia Drossopoulou[1]

[1] Department of Computing, Imperial College London,
180 Queen's Gate, London SW7 2BZ, U.K
[2] Dipartimento di Informatica,
Università del Piemonte Orientale,
Via Bellini 25/G, Alessandria, Italy

Abstract. Object-oriented scripting languages like JavaScript and Python are popular partly because of their dynamic features. These include the runtime modification of objects and classes through addition of fields or updating of methods. These features make static typing difficult and so usually dynamic typing is used. Consequently, errors such as access to non-existent members are not detected until runtime.

We first develop a formalism for an object based language, JS_0, with features from JavaScript, including dynamic addition of fields and updating of methods. We give an operational semantics and static type system for JS_0 using structural types. Our types allow objects to evolve in a controlled manner by classifying members as *definite* or *potential*.

We define a type inference algorithm for JS_0 that is sound with respect to the type system. If the type inference algorithm succeeds, then the program is typeable. Therefore, programmers can benefit from the safety offered by the type system, without the need to write explicitly types in their programs.

1 Introduction

The popularity of scripting languages stems from the flexible programming features they support. These include the runtime modification of objects through addition of fields or updating of methods. These features make static typing difficult and so usually dynamic typing is used. Consequently, errors such as access to non-existent members are not detected until runtime, or, as in JavaScript, not detected at all which can result in a web browser reporting an error when viewing a web page containing JavaScript code.

We introduce JS_0, a formalism of JavaScript[16]. JS_0 supports the standard JavaScript flexible features, e.g. functions creating objects, and dynamic addition/reassignment of fields and methods. We also introduce $\mathrm{JS}_0^{\mathsf{T}}$, an explicitly typed version of JS_0. Types in $\mathrm{JS}_0^{\mathsf{T}}$comprise object types, function types, and

[⋆] Work partly supported by EU within the FET - Global Computing initiative, project DART IST-2001-33477,MURST Cofin'02 project McTati, and MIUR Prin'04 project EOS

Int (the type of integers). Object types list the methods and fields present in the object, $\mu\ \alpha.[\mathtt{m}_1 : (\mathtt{t}_1, \psi_1) \cdots \mathtt{m}_n : (\mathtt{t}_n, \psi_n)]$. We use the μ-binder to allow a type to refer to itself. Our type system permits objects to evolve in a controlled manner by allowing members to be added to an object after it has been created. This is achieved by annotating with ψ, each member of an object type as either *potential* '$\circ$' or *definite* '$\bullet$'.

Function types have the form, $\mathtt{t} = \mu\ \alpha.(\mathsf{O} \times \mathtt{t}_1 \rightarrow \mathtt{t}_2)$, where O is the type of the receiver, $\mathtt{t}_1$ is the type of the formal parameter and $\mathtt{t}_2$ is the return type. As for object types, the bound variable α allows references to $\mathtt{t}$ within O, $\mathtt{t}_1$, and $\mathtt{t}_2$. Thus, $\mu\ \alpha.(\_ \times \_ \rightarrow \alpha)$ is a function that returns a value of the same type as the function itself.

A function can be used as a *global function* if its type does not make any requirements of its receiver. The type system is rich enough to allow typing of many JavaScript programs, and at the same time prevents runtime errors such as access to non-existing members of objects.

We develop a sound type inference algorithm to automatically translate JS_0 code to $\mathrm{JS}_0^{\mathsf{T}}$ code. The algorithm uses *type variables* which represent the type of expressions. Constraints are generated between the type variables. If there is a solution to the constraints this can be used to translate code from JS_0 to $\mathrm{JS}_0^{\mathsf{T}}$. We define a translation between constraints and types that provides the types for the typed version of the code.

In [6] we introduced the language JS_0 and its type system. In this paper we have simplified the presentation of $\mathrm{JS}_0^{\mathsf{T}}$ and its type system and defined a sound type inference algorithm.

This paper is organized as follows. In Section 2 we define the syntax of JS_0 and its operational semantics, and in Section 3 we give $\mathrm{JS}_0^{\mathsf{T}}$. Properties of the type system for $\mathrm{JS}_0^{\mathsf{T}}$ are outlined in Section 4. In Section 5 we show type inference for JS_0, and in Section 6 we show how to turn constraints into types. In Section 7 we compare our work with others. In Section 8 we draw conclusions and outline our future directions. The proofs and a prototype implementation can be found at `http://www.binarylord.com/work/js0/`.

2 JS$_0$

We have developed JS_0 a subset of JavaScript. Figure 1 gives an example JS_0 program that describes an implementation of the JavaScript Date object[1]. We define functions `Date` and `addFn`. The code preceded by the comment `//Main` is the entry point into the program. Although the syntax of JS_0 requires all code to be within a function body to aid presentation we allow a main body of code and the declaration of local variables `x` and `y`. The example demonstrates the *core* JavaScript features we have included:

[1] For more information on the `Date` object see [15]. We give a simplified version and allow the adding of one date to another, with `add`.

```
 1 function Date(x) {
 2  this.mSec = x;
 3  this.add = addFn;
 4  this
 5 }
 6 function addFn(x) {
 7  this.mSec = this.mSec + x.mSec; this
 8 }
 9 //Main
10 x = new Date(1000);
11 y = new Date(100);
12 x.add(y);
```

Fig. 1. Untyped JS$_0$ Date Example

1. creating objects using functions (line 10 and 11),
2. implicit creation of members in objects through assignment (lines 2 and 3), and
3. acquiring methods through assignment of a function to a member (line 3).

We chose these features because, (1) represents the way objects are created in JavaScript, (2) and (3) represent the way objects acquire fields and methods thus giving flexibility to the programmer. JS$_0$ does not include the following JavaScript features: member names as strings, functions as expressions, dynamic removal of members, automatic conversions, and delegation. We omitted the first three as we believe they are not essential in supporting flexible object-oriented programming. The last two while useful can complicate static typing and type inference. We can write the introductory examples from [15] in JS$_0$ assuming libraries of functions, and predefined types *e.g.* floats, strings, etc.

The syntax of JS$_0$ is given in Figure 2. Note that, in the syntax of JS$_0$ we omitted conditional expressions, which were present in [6]. Their presence does not produce conceptual difficulties regarding the type system and type inference. A program is a sequence of function declarations. In JS$_0$ functions may have only one formal parameter. The extension to functions with multiple parameters is trivial, whereas going to a variable number of parameters, as in JavaScript, is an interesting possible future extension.

For a program P, we use $\mathsf{P(f)}$ as a shorthand for looking up the definition of function f in P.

2.1 Operational Semantics

We give a structural operational semantics for JS$_0$ that rewrites tuples of expressions, heaps and stacks into tuples of values, heaps and stacks in the context of a program. The signature of the rewriting relation $\twoheadrightarrow$ is:

$$\twoheadrightarrow : \ \mathit{Program} \ \rightarrow \ \mathit{Exp} \ \times \ \mathit{Heap} \ \times \ \mathit{Stack} \rightarrow \ (\mathit{Val} \ \cup \mathit{Dev}) \times \ \mathit{Heap} \times \ \mathit{Stack}$$

$$
\begin{array}{lll}
\mathsf{P} \in \mathit{Program} & ::= \mathsf{F}^* & \\
\mathsf{F} \in \mathit{FuncDecl} & ::= \texttt{function}\ \mathsf{f}\ (\texttt{x})\ \{\ \texttt{e}\ \} & \\
\mathsf{e} \in \mathit{Exp} & ::= \mathsf{var} & \text{locals} \\
 & \quad \mathsf{f} & \text{function identifier} \\
 & \quad \texttt{new}\ \mathsf{f(e)} & \text{object creation} \\
 & \quad \mathsf{e;\ e} & \text{sequence} \\
 & \quad \mathsf{e.m(e)} & \text{member call} \\
 & \quad \mathsf{e.m} & \text{member select} \\
 & \quad \mathsf{f(e)} & \text{global call} \\
 & \quad \mathsf{lhs} = \mathsf{e} & \text{assignment} \\
 & \quad \texttt{null} & \text{null} \\
 & \quad \texttt{n} & \text{integer} \\
\mathsf{var} \in \mathit{EnvVars} & ::= \texttt{this} \mid \texttt{x} & \\
\mathsf{lhs} \in \mathit{LeftHandSide} & ::= \texttt{x} \mid \mathsf{e.m} &
\end{array}
$$

Identifiers

$$
\begin{array}{ll}
\mathsf{f} \in \mathit{FuncID} & ::= \mathsf{f} \mid \mathsf{f}' \mid \ldots \\
\mathsf{m} \in \mathit{MemberID} & ::= \mathsf{m} \mid \mathsf{m}' \mid \ldots
\end{array}
$$

Fig. 2. Syntax of JS_0

where:

$$
\begin{array}{lllll}
\mathsf{H} & \in & \mathit{Heap} & = \mathit{Addr} \rightarrow_{\text{fin}} \mathit{Obj} & \\
\chi & \in & \mathit{Stack} & = \{\texttt{this}, \texttt{x}\} \rightarrow \mathit{Val} & \text{such that} \quad \chi(\texttt{this}) \in \mathit{Addr} \\
\mathsf{v} & \in & \mathit{Val} & = \{\texttt{null}\} \cup \mathit{FuncID} \cup \mathit{Addr} \cup \mathit{Int} & \\
\mathsf{dv} & \in & \mathit{Dev} & = \{\texttt{nullPntrExc}, \texttt{stuckErr}\} & \\
\mathsf{o} & \in & \mathit{Obj} & = \mathit{MemberID} \rightarrow_{\text{fin}} \mathit{Val} &
\end{array}
$$

The heap maps addresses to objects, where addresses, *Addr*, are $\iota_0, ..\iota_n...$ We use $\rightarrow_{\text{fin}}$ to indicate a finite mapping. As usual, the notation $\mathsf{f}[\texttt{x} \mapsto \texttt{y}]$ denotes updating function f to map $\texttt{x}$ to $\texttt{y}$. Thus, the meaning of heap update $\mathsf{H}[\iota \mapsto \mathsf{v}]$ and stack update $\chi[\texttt{x} \mapsto \mathsf{v}]$ is clear. The stack maps `this` to an address and `x` to a value, where values, *Val*, are function identifiers (denoting functions), addresses (denoting objects), `null` , or integers. Finally objects are finite mappings from member identifiers to values. With $\ll \mathsf{m}_1 : \mathsf{v}_1 ... \mathsf{m}_n : \mathsf{v}_n \gg$ we denote the object mapping m_i to v_i for $i \in 1 \cdots n$.

A full description of the rules is given in [6]. In JavaScript access to non-existent members result in an undefined value not a runtime error[2]. This may cause errors later on in the code. We consider accesses to non-existent members a runtime error and our type system prevents them. Below we give two of the

[2] For some interesting insights into issues surrounding JavaScript's treatment of undefined members see [24].

more interesting rules, (*memAdd*) for adding/updating members and (*memCall*) for calling methods:

$$\frac{\begin{array}{l}\mathsf{e}_1, \mathsf{H}, \chi \twoheadrightarrow \iota, \mathsf{H}_1, \chi_1 \\ \mathsf{e}_2, \mathsf{H}_1, \chi_1 \twoheadrightarrow \mathsf{v}, \mathsf{H}_2, \chi' \\ \mathsf{H}' = \mathsf{H}_2[\iota \mapsto \mathsf{H}_2(\iota)[\mathsf{m} \mapsto \mathsf{v}]]\end{array}}{\mathsf{e}_1.\mathsf{m} = \mathsf{e}_2, \mathsf{H}, \chi \twoheadrightarrow \mathsf{v}, \mathsf{H}', \chi'}\ (\mathit{memAdd}) \qquad \frac{\begin{array}{l}\mathsf{e}_1, \mathsf{H}, \chi \twoheadrightarrow \iota, \mathsf{H}_1, \chi_1 \\ \mathsf{e}_2, \mathsf{H}_1, \chi_1 \twoheadrightarrow \mathsf{v}', \mathsf{H}_2, \chi' \\ \mathsf{H}_2(\iota)(\mathsf{m}) = \mathsf{f} \\ \mathsf{P}(\mathsf{f}) = \mathtt{function}\ \mathsf{f}(\mathtt{x})\ \{\mathsf{e}'\} \\ \chi_2 = \{\mathtt{this} \mapsto \iota, \mathtt{x} \mapsto \mathsf{v}'\} \\ \mathsf{e}', \mathsf{H}_2, \chi_2 \twoheadrightarrow \mathsf{v}, \mathsf{H}', \chi''\end{array}}{\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2), \mathsf{H}, \chi \twoheadrightarrow \mathsf{v}, \mathsf{H}', \chi'}\ (\mathit{memCall})$$

In rule (*memAdd*) we express how objects obtain new members. We first evaluate the receiver, then the right hand side. Using heap update we add/update member m in the receiver. Returning to the example in Figure 1, executing `this.mSec = x` from `Date` with $\chi_0(\mathtt{this}) = \iota_0, \chi_0(\mathtt{x}) = 1000, \mathsf{H}_0(\iota_0) = \ll\gg$, will produce H_1 with $\mathsf{H}_1(\iota_0) = \ll\mathtt{mSec} : 1000\gg$

In rule (*memCall*) we first evaluate the receiver and then the actual parameter of the method. We obtain the function definition (corresponding to the method) by looking up the value of member m in the receiver (obtained by evaluation of e) in P[3]. We execute the body with a stack in which `this` refers to the receiver of the call and `x` to the value of the actual parameter.

For example, executing the code in Figure 1 in the presence of an empty heap, H_0 and χ_0, mapping `x` and `y` to `null` will result in stack $\chi_1(\mathtt{x}) = \iota_0, \chi_1(\mathtt{y}) = \iota_1$ and updated heap H_1, $\mathsf{H}_1(\iota_0) = \ll\mathtt{mSec} : 1100, \mathtt{add} : \mathtt{addFn}\gg, \mathsf{H}_1(\iota_1) = \ll\mathtt{mSec} : 100, \mathtt{add} : \mathtt{addFn}\gg$. For demonstration purposes the stack contains an extra variable `y` although the definition of stack allows only `this` and `x`.

Note that member `add` of both ι_0 and ι_1 has value `addFn`. This indicates that it is an alias of function `addFn`, which is invoked when `x.add(y)` is executed.

3 A Type System for JS$_0$

In this section we introduce JS$_0^\mathsf{T}$ a typed version of JS$_0$. Figure 3 shows the parts of JS$_0^\mathsf{T}$ that differ from JS$_0$ along with the definitions of types. Observe that functions are now annotated with a function type G.

Types $\mathtt{t}_1$, ..., $\mathtt{t}_n$, comprise object types, function types, or Int (the type of integers). Object types list the methods and fields present in the object. We use the μ-binder to allow a type to refer to itself. So $\mu\ \alpha.\mathsf{M}$ where $\mathsf{M} = [\mathsf{m}_1 : (\mathtt{t}_1, \psi_1) \cdots \mathsf{m}_\mathtt{n} : (\mathtt{t}_\mathtt{n}, \psi_\mathtt{n})]$, is the type of an object with members m_1, ..., m_n of type $\mathtt{t}_1$, ..., $\mathtt{t}_n$, respectively. Figure 4 gives a JS$_0^\mathsf{T}$ version of the `Date` example from Figure 1. We use $\mathtt{t}_1$ for type $[\mathtt{mSec} : (\mathsf{Int}, \circ),\ \mathtt{add} : ((\mathtt{t}_2 \times \mathtt{t}_2 \to \mathtt{t}_2), \circ)]$ and $\mathtt{t}_2$ for type $\mu\ \alpha.[\mathtt{mSec} : (\mathsf{Int}, \bullet),\ \mathtt{add} : ((\alpha \times \alpha \to \alpha), \bullet)]$. To aid the presentation we allow local variable type declarations on lines 10 and 11. These are not part of the syntax of JS$_0^\mathsf{T}$, where type declarations are only allowed for the

[3] For clearness of presentation we omit P from the reduction rules.

Syntax

$$\begin{array}{lll}
\mathsf{P} \in \textit{Program} & ::= & \mathsf{F}^* \\
\mathsf{F} \in \textit{FuncDecl} & ::= & \texttt{function}\ \mathsf{f(x)} : \mathsf{G}\ \{\ \mathsf{e}\}
\end{array}$$

Types

$$\begin{array}{lll}
\mathsf{t} \in \textit{Type} & ::= & \mathsf{O} \mid \mathsf{G} \mid \mathsf{Int} \\
\mathsf{tp} \in \textit{PreType} & ::= & \alpha \mid \mathsf{t} \\
\mathsf{O} \in \textit{ObjType} & ::= & \mu\ \alpha.\mathsf{M} \mid \mathsf{M} \\
\mathsf{G} \in \textit{FuncType} & ::= & \mu\ \alpha.\mathsf{R} \mid \mathsf{R} \\
\mathsf{M} \in \textit{ObjMembers} & ::= & [(\mathsf{m} : \mathsf{tm})^*] \\
\mathsf{tm} \in \textit{MemberType} & ::= & (\mathsf{tp}, \psi) \\
\mathsf{R} \in \textit{FuncRow} & ::= & (\mathsf{O} \times \mathsf{tp} \rightarrow \mathsf{tp}) \\
 & & \\
\psi \in \textit{Annotation} & ::= & \circ \mid \bullet \\
\alpha \in \textit{ObjVar} & ::= & \alpha \mid \alpha' \mid \alpha'' \ \ldots
\end{array}$$

Fig. 3. Syntax of $\mathrm{JS}_0^{\mathsf{T}}$

parameter of a method and `this` and are implicitly given in the function type for a function.

```
function Date(x):(t₁ × Int → t₂) {
 this.mSec = x;
 this.add = addFn;
 this
}
function addFn(x):(t₂ × t₂ → t₂) {
 this.mSec = this.mSec + x.mSec; this;
}
//Main
t₂ x = new Date(1000);
t₂ y = new Date(100);
x.add(y);
```

Fig. 4. Typed JS_0 Date Example.

Our type system permits objects to evolve in a controlled manner by allowing members to be added to an object after it has been created. This is achieved by annotating each member of an object type as either *potential* '$\circ$' or *definite* '$\bullet$' e.g. $\mathsf{mSec} : (\mathsf{Int}, \circ)$ in t_1 and $\mathsf{mSec} : (\mathsf{Int}, \bullet)$ in t_2. When a potential member is assigned to, it becomes definite, replacing $\circ$ with $\bullet$. To keep the type system manageable we only track assignments to variables (formal parameters and `this`) within the scope of a function. In a well-typed program potential members may not be accessed until they have been assigned to.

Function types, $(\mathsf{O} \times \mathsf{t}_1 \to \mathsf{t}_2)$ or $\mu\ \alpha.(\mathsf{O} \times \mathsf{t}_1 \to \mathsf{t}_2)$, list the type of the receiver, O, which is an object type, the type of the parameter, t_1, and the type of the return value of the function, t_2. As for object types the μ-binder allows a function type to refer to itself, thus $\mu\ \alpha.(\_ \times \_ \to \alpha)$ is a function that returns a function with its type.

If the type of m is an object type, or Int, the member represents a field. If the type of m is a function type, then m represents a method. In case the type of the m is α then if α is bound in an objects type the member is a field, whereas if it is bound in a function type it is a method. An object type is well-formed if it is closed and contains unique member definitions that are themselves well-formed. A function type $\mathsf{G} = \mu\ \alpha.\mathsf{R}$ (or $\mathsf{G} = \mathsf{R}$) is well-formed, $\vdash \mathsf{G}\ \diamond$, if the receiver, parameter and return types of $\mathsf{G}[\alpha/\mathsf{R}]$ (or R) are well-formed.

For a well-formed object type O, define $\mathsf{O}(\mathsf{m})$, which selects the annotated type of the member m in O (if it is defined) by first defining selection from $\mathsf{O} = [\mathsf{m}_1 : (\mathsf{t}_1, \psi_1) \cdots \mathsf{m}_n : (\mathsf{t}_n, \psi_n)]$ as

$$\mathsf{O}(\mathsf{m}) = \begin{cases} (\mathsf{t}_i, \psi_i) & \text{if } \mathsf{m} = \mathsf{m}_i \text{ for some } i,\ 1 \le i \le n \\ \mathit{Udf} & \text{otherwise} \end{cases}$$

and then if $\mathsf{O} = \mu\ \alpha.\mathsf{M}$,

$$\mathsf{O}(\mathsf{m}) = \mathsf{M}[\alpha/\mathsf{O}](\mathsf{m})$$

That is, the type is closed by substituting occurrences of α with the enclosing type. Therefore, if O is well-formed, then also $\mathsf{O}(\mathsf{m})$ is well-formed.

With $\mathsf{O}[\mathsf{m} \mapsto (\mathsf{t}, \psi)]$ we denote the *updating of the member* m *to type* t *with annotation* ψ *in* O. Note that, if O and t are well-formed, then $\mathsf{O}[\mathsf{m} \mapsto (\mathsf{t}, \psi)]$ is well-formed.

Congruence and Subtyping *Congruence* between types is defined in Figure 5. With $\mathsf{t}_1[\alpha/\mathsf{t}_2]$, we denote the substitution of the free occurrences of α in t_1 with t_2. Object types are congruent up to α-conversion, permutation of their members, and unfolding of the bound variable, and function types are congruent up to α-conversion, and unfolding of the bound variable.

The *subtyping* judgement $\mathsf{t} \le \mathsf{t}'$, defined in Figure 6, means that an object or function of type t can be used whenever one of type t' is required. For object types we have subtyping in width. If $\mathsf{O} \le \mathsf{O}'$, then all definite members of O' must be present and congruent with those in O, and all potential members of O' must be present as potential or definite members of O with congruent types. This condition is needed to insure that the addition of a new member to an object does not break compatibility.

Returning to the example in Figure 4 we see that t_2 is a subtype of t_1 because all members of t_1 are also members of t_2, and have congruent types; furthermore, all members of t_2 are definite.

Reflexivity

$$\frac{}{\mathsf{t} \equiv \mathsf{t}}$$

Unfolding

$$\frac{}{\begin{array}{l}\mu\ \alpha.\mathsf{M} \equiv \mathsf{M}[\alpha/\mu\ \alpha.\mathsf{M}] \\ \mu\ \alpha.\mathsf{R} \equiv \mathsf{R}[\alpha/\mu\ \alpha.\mathsf{R}]\end{array}}$$

Transitivity

$$\frac{\mathsf{t}_1 \equiv \mathsf{t}_2 \quad \mathsf{t}_2 \equiv \mathsf{t}_3}{\mathsf{t}_1 \equiv\ \mathsf{t}_3}$$

Alpha – conversion

$$\frac{\alpha' \notin\ \mathcal{FV}(\mathsf{M})}{\mu\ \alpha.\mathsf{M} \equiv \mu\ \alpha'.\mathsf{M}[\alpha/\alpha']} \qquad \frac{\alpha' \notin\ \mathcal{FV}(\mathsf{R})}{\mu\ \alpha.\mathsf{R} \equiv \mu\ \alpha'.\mathsf{R}[\alpha/\alpha']}$$

Reordering

$$\frac{\forall\ \mathsf{m}\ :\ \mathsf{M}(\mathsf{m})\ \equiv\ \mathsf{M}'(\mathsf{m})}{\mathsf{M}\ \equiv\ \mathsf{M}'}$$

Functions

$$\frac{\mathsf{M}\ \equiv\ \mathsf{M}'\quad \mathsf{t}_1\ \equiv\ \mathsf{t}_1'\quad \mathsf{t}_2\ \equiv\ \mathsf{t}_2'}{(\mathsf{M} \times \mathsf{t}_1 \rightarrow \mathsf{t}_2)\ \equiv\ (\mathsf{M}' \times \mathsf{t}_1' \rightarrow \mathsf{t}_2')}$$

Members

$$\frac{\mathsf{t}\ \equiv\ \mathsf{t}'}{(\mathsf{t}, \psi) \equiv\ (\mathsf{t}', \psi)}$$

Fig. 5. Congruence for types

For function types subtyping coincides with congruence. In future versions of this work we may relax this restriction and allow contravariance of the receiver and parameter type and covariance of the return type. However, since type inference was our main aim, we started with the reduced system. Given types t and t′ it is decidable whether t ≤ t′ or not.

$$\frac{\psi' = \bullet \quad \Longrightarrow \quad \psi = \bullet}{\psi \leq \psi'} \qquad \frac{\begin{array}{c}\mathsf{t} \equiv \mathsf{t}' \\ \psi \leq \psi'\end{array}}{(\mathsf{t}, \psi) \leq (\mathsf{t}', \psi')} \qquad \frac{\mathsf{t} \equiv\ \mathsf{t}'}{\mathsf{t} \leq \mathsf{t}'}$$

$$\frac{\forall\ \mathsf{m}\ :\ \mathsf{O}'(\mathsf{m}) = (\mathsf{t}', \psi') \quad \Longrightarrow \quad (\mathsf{O}(\mathsf{m}) = (\mathsf{t}, \psi)\ \wedge\ \ (\mathsf{t}, \psi) \leq (\mathsf{t}', \psi'))}{\mathsf{O} \leq \mathsf{O}'}$$

Fig. 6. Subtyping

3.1 Typing Expressions

Typing expression e in the context of program P, and environment Γ has form:

$$\mathsf{P}, \Gamma \vdash \mathsf{e} : \mathsf{t} \parallel \Gamma'$$

The environment, $\Gamma = \{\texttt{this} : \mathsf{O}, \texttt{x} : \mathsf{t}\}$, maps the receiver, `this`, to a well-formed object type, and the formal parameter, `x`, to a well-formed type. The environment on the right hand side of the judgement, Γ', reflects the changes to the type of the receiver or parameter while typing the expression. The only possible difference between Γ and Γ' is that some members that are annotated with $\circ$ in Γ are annotated with $\bullet$ in Γ'. With $\Gamma[\mathsf{var} \mapsto \mathsf{t}]$ we denote the *updating of* var *to type* t *in* Γ.

Consider the typing rules of Figure 7. Rules (*var*), (*func*), (*const*), and (*seq*) are straightforward. Note that `null` may have any object type.

In rule (*memAcc*) the expression e must be of an object type in which the member m is definite, i.e. with annotation $\bullet$.

We use the notation $\mathsf{G}(\mathtt{this})$, $\mathsf{G}(\mathtt{x})$, and $\mathsf{G}(\mathtt{ret})$, to denote the types of the receiver, parameter and return value of G. As for member selection, we define for $\mathsf{G} = (\mathsf{O} \times \mathsf{t}_1 \to \mathsf{t}_2) : \mathsf{G}(\mathtt{this}) = \mathsf{O} \qquad \mathsf{G}(\mathtt{x}) = \mathsf{t}_1 \qquad \mathsf{G}(\mathtt{ret}) = \mathsf{t}_2$ and for $\mathsf{G} = \mu\ \alpha.\mathsf{R}$, we define $\mathsf{G}(\mathtt{z}) = (\mathsf{R}[\alpha/\mathsf{G}])(\mathtt{z})$ where $\mathtt{z} \in \{\mathtt{x}, \mathtt{this}, \mathtt{ret}\}$.

Rule (*methCall*) checks that the type of the receiver is an object type in which the member m has a definite function type. Moreover, the type of the receiver and actual parameter must be subtypes of the declared type of the receiver and formal parameter.

In (*call*) we consider global calls and constructors, and require that the type of the receiver defined in the function has no definite members. This is consistent with the operational semantics, as in the case of global call and object creation we start with an empty receiver object.

In rule (*assignAdd*) in Γ'' we ensure that member m (of `this` or the formal parameter) is definite. From this point onwards, member m of var may be accessed. For example, consider the expression $\mathtt{x}.\mathsf{m}_2 = \mathtt{x}$ in the environment Γ, where $\Gamma(\mathtt{x})$ has type $\mathsf{t} = \mu\ \alpha.[\mathsf{m}_1 : (\mathsf{Int}, \bullet),\ \mathsf{m}_2 : (\alpha, \circ)]$. The expression is well-typed in Γ and we have $\mathsf{P}, \Gamma \vdash \mathtt{x}.\mathsf{m}_2 = \mathtt{x} : \mathsf{t} \parallel \Gamma'$ where Γ' maps `this` to $\Gamma(\mathtt{this})$ and $\mathtt{x}$ to $[\mathsf{m}_1 : (\mathsf{Int}, \bullet), \mathsf{m}_2 : (\mathsf{t}, \bullet)]$. This reflects the updating of member m_2. Any aliases to `this` or the formal parameter will not *see* the update of a member. This would require dataflow analysis techniques and is beyond the scope of this work. The fact that the type system requires a member to be *known* (either as potential or definite) for an assignment to succeed is not a limitation. The process of type inference will find all members for a type with their appropriate type and annotation.

Rule (*assignUpd*) is used when the assignment is to a definite member m. In this case we just check that the type of the expression on the right hand side is a subtype of the type of the member m.

A program P is *well-formed* if all the function declarations in P are well-typed. Figure 7 gives the definition.

4 Formal Properties of the Type System

In this section we give the relevant definitions and the statement that asserts that our type system is sound w.r.t. to the operational semantics given in Section

Typing Expressions

$$\frac{}{\begin{array}{l}\mathsf{P},\Gamma\vdash \mathtt{this}:\Gamma(\mathtt{this})\parallel\Gamma\\ \mathsf{P},\Gamma\vdash \mathtt{x}:\Gamma(\mathtt{x})\parallel\Gamma\end{array}}\,(var) \qquad \frac{\mathsf{P}(\mathsf{f})=\mathtt{function}\ \mathsf{f}(\mathtt{x}):\mathsf{G}...}{\mathsf{P},\Gamma\vdash \mathsf{f}:\mathsf{G}\parallel\Gamma}\,(func)$$

$$\frac{}{\begin{array}{l}\mathsf{P},\Gamma\vdash \mathtt{null}\ :\mathsf{O}\parallel\Gamma\\ \mathsf{P},\Gamma\vdash \mathtt{n}:\mathsf{Int}\parallel\Gamma\end{array}}\,(const) \qquad \frac{\begin{array}{l}\mathsf{P},\Gamma\vdash \mathsf{e}_1:\mathsf{t}\parallel\Gamma'\\ \mathsf{P},\Gamma'\vdash \mathsf{e}_2:\mathsf{t}'\parallel\Gamma''\end{array}}{\mathsf{P},\Gamma\vdash \mathsf{e}_1;\ \mathsf{e}_2:\mathsf{t}'\parallel\Gamma''}\,(seq)$$

$$\frac{\begin{array}{l}\mathsf{P},\Gamma\vdash \mathsf{e}:\mathsf{O}\parallel\Gamma'\\ \mathsf{O}(\mathsf{m})=(\mathsf{t}',\bullet)\end{array}}{\mathsf{P},\Gamma\vdash \mathsf{e.m}:\mathsf{t}'\parallel\Gamma'}\,(memAcc) \qquad \frac{\begin{array}{l}\mathsf{P},\Gamma\vdash \mathsf{e}:\mathsf{t}\parallel\Gamma'\\ \mathsf{t}\le\Gamma'(\mathtt{x})\end{array}}{\mathsf{P},\Gamma\vdash \mathtt{x}=\mathsf{e}:\mathsf{t}\parallel\Gamma'}\,(varAss)$$

$$\frac{\begin{array}{l}\mathsf{P},\Gamma\vdash \mathsf{e}_1:\mathsf{O}\parallel\Gamma'\\ \mathsf{O}(\mathsf{m})=(\mathsf{G},\bullet)\\ \mathsf{P},\Gamma'\vdash \mathsf{e}_2:\mathsf{t}'\parallel\Gamma''\\ \mathsf{t}'\le\mathsf{G}(\mathtt{x})\\ \mathsf{O}\le\mathsf{G}(\mathtt{this})\end{array}}{\mathsf{P},\Gamma\vdash \mathsf{e}_1.\mathsf{m}(\mathsf{e}_2):\mathsf{G}(\mathtt{ret})\parallel\Gamma''}\,(methCall) \qquad \frac{\begin{array}{l}\mathsf{P},\Gamma\vdash \mathsf{e}:\mathsf{t}\parallel\Gamma'\\ \mathsf{P}(\mathsf{f})=\mathtt{function}\ \mathsf{f}(\mathtt{x}):\mathsf{G}...\\ \mathsf{t}\le\mathsf{G}(\mathtt{x})\\ \{\mathsf{t}'\mid(\mathsf{G}(\mathtt{this}))(\mathsf{m})=(\mathsf{t}',\bullet)\}=\emptyset\end{array}}{\begin{array}{l}\mathsf{P},\Gamma\vdash \mathtt{new}\ \mathsf{f}(\mathsf{e}):\mathsf{G}(\mathtt{ret})\parallel\Gamma'\\ \mathsf{P},\Gamma\vdash \mathsf{f}(\mathsf{e}):\mathsf{G}(\mathtt{ret})\parallel\Gamma'\end{array}}\,(call)$$

$$\frac{\begin{array}{l}\mathsf{P},\Gamma\vdash \mathsf{e}_2:\mathsf{t}\parallel\Gamma'\\ \Gamma'(\mathsf{var})\ =\mathsf{O}\\ \mathsf{O}(\mathsf{m})=(\mathsf{t}'',\psi)\\ \mathsf{t}\le\mathsf{t}''\\ \Gamma''=\Gamma'[\mathsf{var}\mapsto\mathsf{O}[\mathsf{m}\mapsto(\mathsf{t}'',\bullet)]]\end{array}}{\mathsf{P},\Gamma\vdash \mathsf{var.m}=\mathsf{e}_2:\mathsf{t}\parallel\Gamma''}\,(assignAdd) \qquad \frac{\begin{array}{l}\mathsf{P},\Gamma\vdash \mathsf{e}_1:\mathsf{O}\parallel\Gamma'\\ \mathsf{P},\Gamma'\vdash \mathsf{e}_2:\mathsf{t}\parallel\Gamma''\\ \mathsf{O}(\mathsf{m})=(\mathsf{t}'',\bullet)\\ \mathsf{t}\le\mathsf{t}''\end{array}}{\mathsf{P},\Gamma\vdash \mathsf{e}_1.\mathsf{m}=\mathsf{e}_2:\mathsf{t}\parallel\Gamma''}\,(assignUpd)$$

Well – formed Programs

$$\frac{\begin{array}{l}\forall\ \mathsf{f}\ :\quad \mathsf{P}(\mathsf{f})=\mathtt{function}\ \mathsf{f}(\mathtt{x}):\mathsf{G}\ \{\mathsf{e}\}\ \wedge\ \vdash\mathsf{G}\diamond\\ \qquad\qquad\Longrightarrow\ \mathsf{P},\{\ \mathtt{this}:\mathsf{G}(\mathtt{this}),\mathtt{x}:\mathsf{G}(\mathtt{x})\ \}\vdash \mathsf{e}:\mathsf{t}\parallel\Gamma''\ \wedge\ \mathsf{t}\le\mathsf{G}(\mathtt{ret})\end{array}}{\vdash\mathsf{P}\diamond}$$

Fig. 7. Type Rules for Expressions in $\mathrm{JS}_0^{\mathsf{T}}$

2.1. We assume that types are well-formed. We first define the notion of a value being compatible with a given type. The definition is given co-inductively by first defining the properties that any agreement relation between values and well-formed types should have.

Definition 1. *Given a heap,* H*, and a program,* P*, we say that* $\mathsf{A} \subseteq (\mathit{Val} \times \mathit{Type})$ *is an* agreement relation *if the following conditions are satisfied:*

- *if* $(\mathtt{null}, \mathsf{t}) \in \mathsf{A}$*, then* $\mathsf{t} = \mathsf{O}$ *for some well-formed* O*,*
- *if* $(\mathtt{n}, \mathsf{t}) \in \mathsf{A}$*, then* $\mathsf{t} = \mathsf{Int}$*,*
- *if* $(\mathsf{f}, \mathsf{t}) \in \mathsf{A}$*, then* $\mathsf{P}(\mathsf{f}) = \mathtt{function}\ \mathsf{f}(\mathtt{x}) : \mathsf{G}$ *and* $\mathsf{G} \equiv \mathsf{t}$*,*
- *if* $(\iota, \mathsf{t}) \in \mathsf{A}$*, then*
  $\mathsf{t} = \mathsf{O}$ *for some well-formed* O*,* $\mathsf{H}(\iota) = \ll \mathsf{m}_1 : \mathsf{v}_1 \dots \mathsf{m}_p : \mathsf{v}_p \gg$*, and*
  - $\mathsf{O}(\mathsf{m}) = (\mathsf{t}', \bullet) \implies \mathsf{m} = \mathsf{m}_i$ *for some* i*,* $i \in 1...p$*, and* $(\mathsf{v}_i, \mathsf{t}') \in \mathsf{A}$
  - $\mathsf{O}(\mathsf{m}) = (\mathsf{t}', \circ)$ *and* $\mathsf{m} = \mathsf{m}_i$ *for some* i*,* $i \in 1...p$*,* $\implies (\mathsf{v}_i, \mathsf{t}') \in \mathsf{A}$

If A and A' are agreement relations, then $\mathsf{A} \cup \mathsf{A}'$ is also an agreement relation. Therefore, the union of all agreement relations defines a relation between values and types, which determines when a value has a given type.

Definition 2. Value v is compatible with type t in H

$$\mathsf{P}, \mathsf{H} \vdash \mathsf{v} \blacktriangleleft \mathsf{t}$$

if $(\mathsf{v}, \mathsf{t}) \in \mathsf{A}$ *for some agreement relation* A *on* H *and* P*.*

Note that an address may be compatible with more than one type. In particular, a value compatible with a type is compatible with all its supertypes.

Lemma 1. *If* $\mathsf{t} \leq \mathsf{t}'$ *and* $\mathsf{P}, \mathsf{H} \vdash \mathsf{v} \blacktriangleleft \mathsf{t}$ *then* $\mathsf{P}, \mathsf{H} \vdash \mathsf{v} \blacktriangleleft \mathsf{t}'$*.*

In the following we define when *a stack* χ *and a heap* H *are compatible with an environment* Γ.

Definition 3. $\mathsf{P}, \Gamma \vdash \mathsf{H}, \chi \diamond$ *holds if* $\mathsf{P}, \mathsf{H} \vdash \chi(\mathtt{this}) \blacktriangleleft \Gamma(\mathtt{this})$ *and* $\mathsf{P}, \mathsf{H} \vdash \chi(\mathtt{x}) \blacktriangleleft \Gamma(\mathtt{x})$*.*

We can now state the Soundness Theorem. The theorem asserts that if an expression is well-typed,

$$\mathsf{P}, \Gamma \vdash \mathsf{e} : \mathsf{t} \parallel \Gamma'$$

then the evaluation of the expression starting in a heap and stack that are compatible with Γ will not get stuck. That is, the result of the evaluation is either a value compatible with type t, or it is a `nullPntrExc` exception. In particular, it is not a `stuckErr` error. Moreover, the stack and heap produced are compatible with the final environment Γ'.

Theorem 1. [*Type Soundness*] *For a well-formed program* P*, environment* Γ*, and expression* e*, such that:*

$$\mathsf{P}, \Gamma \vdash \mathsf{e} : \mathsf{t} \parallel \Gamma'$$

If $\mathsf{P}, \Gamma \vdash \mathsf{H}, \chi \diamond$ *and* $\mathsf{e}, \mathsf{H}, \chi \twoheadrightarrow \mathsf{w}, \mathsf{H}', \chi'$*, then either*

- $\mathsf{w} = \mathtt{nullPntrExc}$*, or*
- $\mathsf{w} = \mathsf{v}$*,* $\mathsf{P}, \mathsf{H}' \vdash \mathsf{v} \blacktriangleleft \mathsf{t}$*, and* $\mathsf{P}, \Gamma' \vdash \mathsf{H}', \chi' \diamond$*.*

5 Type Inference

We show how type inference for JS_0 can be expressed as a finite system of constraints between type variables. Type variables are used to represent the type of an expression. From a JS_0 program, we can generate a set of type variables with constraints between them. Constraints represent the relationships we expect between types in the program. For example, that the actual parameter to a function call should be a subtype of the formal parameter.

If the constraints have a solution we say that they are satisfiable. A solution can be used to translate a JS_0 program into an equivalent JS_0^{T} program. This involves annotating the JS_0 program with type declarations. We show that the annotated program is well-typed.

5.1 Type Variables

As in [20,3,21,18], we use type variables to express the - yet unknown - types of expressions. Thus, $[\![\texttt{new Date(1000)}]\!]$ expresses the type of `new Date(1000)`.

Because the types of `this` and `x` differ for different occurrences in the same method body, we use labels to distinguish them, for example, $[\![\texttt{this\_1}]\!]$, $[\![\texttt{x\_2}]\!]$, $[\![\texttt{this\_3}]\!]$, *etc.*. Labeled type variables[4] $[\![\texttt{this\_f}]\!]$ and $[\![\texttt{x\_f}]\!]$ represent the type of `this` and `x` at the beginning of the function `f`, and $[\![\texttt{ret\_f}]\!]$ represents the return type of the function.

We generate a new label for each method call; this label is used to generate three type variables. These variables denote the type of the receiver, parameter and return type of the method. For example, for `x.add(y)` we could use label `5` which would generate $[\![\texttt{call\_this\_5}]\!]$, $[\![\texttt{call\_x\_5}]\!]$, and $[\![\texttt{call\_ret\_5}]\!]$. Note that these type variables depend on the label but not on the name of the method.[5] Figure 8 gives the syntax of labeled expressions.

Figure 9 defines type variables. Type variables can be used to describe function types, *e.g.* $(\tau \times \tau' \to \tau'')$, or object types, *e.g.* $[\mathsf{m}{:}(\tau,\psi)]$ with the obvious meaning.

5.2 Constraints and Solutions

A solution, S, is a mapping from type variables to types. For the `Date` example, let $\mathtt{t}_2$ be $\mu\ \alpha.[\texttt{mSec} : (\mathsf{Int}, \bullet),\ \texttt{add} : ((\alpha \times \alpha \to \alpha), \bullet)]$, S_0 represents part of a solution, as follows:

$$\begin{aligned}
\mathsf{S}_0([\![\texttt{this\_Date}]\!]) &= [\texttt{mSec} : (\mathsf{Int}, \circ),\ \texttt{add} : ((\mathtt{t}_2 \times \mathtt{t}_2 \to \mathtt{t}_2), \circ)] \\
\mathsf{S}_0([\![\texttt{this\_1}]\!]) &= [\texttt{mSec} : (\mathsf{Int}, \bullet),\ \texttt{add} : ((\mathtt{t}_2 \times \mathtt{t}_2 \to \mathtt{t}_2), \circ)] \\
\mathsf{S}_0([\![\texttt{ret\_Date}]\!]) &= \mathtt{t}_2 \quad \mathsf{S}_0([\![\texttt{x\_Date}]\!]) = \mathsf{Int} \\
\mathsf{S}_0([\![\texttt{this\_Date.mSec}]\!]) &= \mathsf{Int} \quad \mathsf{S}_0([\![\texttt{this\_5}]\!]) = [\texttt{mSec} : (\mathsf{Int}, \bullet)]
\end{aligned}$$

[4] In [3] Agesen et al. use a similar labeling, $[\![\mathbb{e}]\!]_\tau$, to indicate who the sender, τ, of a method call is.

[5] It is possible to optimize the creation of new variables at the call site, for example by sharing some of them. Refer to Section 7 where we discuss [25] which shows possible optimizations.

$$\begin{array}{lll}
\mathbb{e} \in \mathit{LabExp} & ::= & \mathbb{var} \mid \mathsf{f} \mid \mathtt{new}\ \mathsf{f}(\mathbb{e}) \mid \mathbb{e}; \mathbb{e} \mid \mathbb{e}.\mathsf{m}(\mathbb{e}) \mid \\
 & & \mathbb{e}.\mathsf{m} \mid \mathsf{f}(\mathbb{e}) \mid \mathbb{lhs} = \mathbb{e} \mid \mathtt{null} \mid \mathtt{n} \mid \mathbb{ie} \\
\mathbb{var} \in \mathit{LabEnvVars} & ::= & \mathtt{this\_l} \mid \mathtt{x\_l} \\
\mathbb{lhs} \in \mathit{LeftSide} & ::= & \mathtt{x\_l} \mid \mathbb{e}.\mathsf{m} \\
\mathbb{ie} \in \mathit{InferExp} & ::= & \mathtt{ret\_f} \mid \mathtt{call\_this\_l} \mid \mathtt{call\_x\_l} \mid \mathtt{call\_ret\_l} \\
 & & \\
\mathtt{l} \in \mathit{Lab} & ::= & \mathsf{1} \mid \mathsf{2} \mid \ldots \mid \mathsf{f} \mid \mathsf{f}' \mid \ldots
\end{array}$$

Fig. 8. Syntax of Labeled Expressions

Constraints between type variables express the relationship between the types of expressions, i.e. which members a type must have, how the members of two types may differ and whether a type has any definite members. The syntax of constraints is given in Figure 9. There are three kinds of constraint: $\tau \leq \rho$, $\tau \lhd \tau$, and τ°. We use c to range over constraints and C for a set of constraints.

Figure 10, rule (*solSat*), defines that S satisfies a set of constraints, $\mathsf{S} \vdash \mathsf{C}$, if it satisfies each constraint. We now discuss each kind of constraint and how it is satisfied by a solution.

– $\tau \leq \rho$ - requires a type variable to be a subtype of ρ: Thus, $\tau \leq \mathsf{Int}$ requires τ to be `Int`, *c.f.* rule (*solInt*); while $\tau \leq \tau'$ requires τ to be a subtype of τ', *c.f.* (*solSub*); while $\tau \leq (\tau_1 \times \tau_2 \to \tau_3)$ requires τ to be the function type composed from τ_1, τ_2 and τ_3, *c.f.* (*solSubFunc*); finally, $\tau \leq [\mathsf{m} : (\tau', \psi)]$ requires τ to have a member m of type τ' with annotation at least ψ, *c.f.* (*solMemChange*).
  Thus, $\mathsf{S}_0 \vdash [\![\mathtt{this\_1}]\!] \leq [\![\mathtt{this\_Date}]\!]$, and $\mathsf{S}_0 \vdash [\![\mathtt{this\_1}]\!] \leq [\![\mathtt{this\_5}]\!]$,
  $\mathsf{S}_0 \vdash [\![\mathtt{this\_Date}]\!] \leq [\mathtt{mSec} : ([\![\mathtt{this\_Date.mSec}]\!], \circ)]$, but
  $\mathsf{S}_0 \not\vdash [\![\mathtt{this\_Date}]\!] \leq [\mathtt{mSec} : ([\![\mathtt{this\_Date.mSec}]\!], \bullet)]$.

– $\tau \lhd_{\mathsf{m}} \tau'$ - requires τ and τ' to have the same members with the same types, but member m can be potential in τ' but must be definite in τ, *c.f.* rule (*solMemChange*).
  For example, $\mathsf{S}_0 \vdash [\![\mathtt{this\_1}]\!] \lhd_{\mathtt{mSec}} [\![\mathtt{this\_Date}]\!]$, while
  $\mathsf{S}_0 \not\vdash [\![\mathtt{this\_Date}]\!] \lhd_{\mathtt{mSec}} [\![\mathtt{this\_1}]\!]$. Also $\mathsf{S}_0 \not\vdash [\![\mathtt{this\_1}]\!] \lhd_{\mathtt{mSec}} [\![\mathtt{this\_5}]\!]$. Note, however, that $\mathsf{S}_0 \vdash [\![\mathtt{this\_1}]\!] \leq [\![\mathtt{this\_5}]\!]$ – this should clarify the difference between the two kinds of constraint.

– τ° - requires τ to have no definite members, *c.f.* rule (*solNoDefs*). This is needed for constructors and global functions whose receiver must have no definite members. For example, $\mathsf{S}_0 \vdash [\![\mathtt{this\_Date}]\!]^{\circ}$.

5.3 Constraint Generation

Constraint generation for a JS$_0$ program produces a set of constraints between type variables, and a labeled version of the original expression, $\mathbb{e}$. A

Type Variables

$$\tau \ ::= [\![\mathbb{e}]\!]$$

Constraints

$$\begin{array}{rcl} \rho \ \in \mathit{ConstRhs} & ::= & \tau \ \mid \ \sigma \ \mid \ [\mathsf{m} : (\tau, \psi)] \\ \sigma \in \mathit{FuncInt} & ::= & (\tau \times \tau) \to \ \tau \ \mid \ \mathsf{Int} \\ \\ \mathsf{c} \ \in \ \mathit{Const} & ::= & \tau \ \leq \ \rho \ \mid \ \tau \ \lhd_{\mathsf{m}} \tau \ \mid \ \tau^{\circ} \\ \mathsf{C} \in & & \mathcal{P}(\mathit{Const}) \end{array}$$

Fig. 9. Syntax of Type Variables and Constraints

pre-environment, $\gamma = \{\texttt{this} : \texttt{l},\ \texttt{x} : \texttt{l}',\ \texttt{lab} : \texttt{L}\}$, keeps track of the current labeling of `this` and `x` along with the set of labels used so far, stored in the set `L`. Constraint generation for an expression e in the context of a pre-environment, γ, has the form:

$$\gamma \vdash \mathsf{e} \ : \ \mathbb{e} \ || \ \gamma' \ || \ \mathsf{C}$$

$$\frac{\begin{array}{l}\mathsf{C} = \{\mathsf{c}_1 ... \mathsf{c}_n\} \\ \mathsf{S} \vdash \mathsf{c}_i \ \forall \ i \ \in \ 1...n\end{array}}{\mathsf{S} \vdash \mathsf{C}} \ (\mathit{solSat})$$

$$\frac{\mathsf{S}(\tau) \leq \mathsf{S}(\tau')}{\mathsf{S} \vdash \tau \ \leq \ \tau'} \ (\mathit{solSub}) \qquad \frac{\mathsf{S}(\tau) \leq (\mathsf{S}(\tau_1) \times \mathsf{S}(\tau_2) \to \mathsf{S}(\tau_3))}{\mathsf{S} \vdash \tau \ \leq \ (\tau_1 \times \tau_2 \to \tau_3)} \ (\mathit{solSubFunc})$$

$$\frac{\mathsf{S}(\tau) = \mathsf{Int}}{\mathsf{S} \vdash \tau \ \leq \ \mathsf{Int}} \ (\mathit{solInt}) \qquad \frac{\mathsf{S}(\tau)(\mathsf{m}) \leq (\mathsf{S}(\tau'), \psi)}{\mathsf{S} \vdash \tau \ \leq \ [\mathsf{m} : (\tau', \psi)]} \ (\mathit{solMember})$$

$$\frac{\begin{array}{l}\forall \ \mathsf{m}' \neq \mathsf{m} \ : \ \mathsf{S}(\tau)(\mathsf{m}') \equiv \ \mathsf{S}(\tau')(\mathsf{m}') \\ \mathsf{S}(\tau)(\mathsf{m}) \leq \mathsf{S}(\tau')(\mathsf{m})\end{array}}{\mathsf{S} \vdash \tau \ \lhd_{\mathsf{m}} \tau'} \ (\mathit{solMemChange})$$

$$\frac{\{\mathsf{m} \mid \mathsf{S}(\tau)(\mathsf{m}) = (\mathsf{t}, \bullet)\} = \emptyset}{\mathsf{S} \vdash \tau^{\circ}} \ (\mathit{solNoDefs})$$

Fig. 10. Solution Satisfaction

where γ' reflects the changes to the labeling of `this`, `x` and `lab` while generating constraints. The constraints generated for an expression consist of the union of the constraints for each subexpression augmented by local constraints.

In (*var*) we generate a labeled expression for `this` and `x` by looking in the pre-environment for the current label. No constraints are generated.

In (*funcId*) we require f to have a function type derived from the type of the receiver, parameter and return value of the function. The type variables come from the initial labeled `this` and `x` and the labeled return variable `ret_f`. For example, function identifier `addFn` produces constraint:

$$[\![\mathtt{addFn}]\!] \ \leq \ ([\![\mathtt{this\_addFn}]\!] \times [\![\mathtt{x\_addFn}]\!] \rightarrow [\![\mathtt{ret\_addFn}]\!])$$

In (*assignAdd*) we use var for `this` or `x`, and we model the change of member m of var to definite. $\mathsf{var\_}\mathtt{l}$ and $\mathsf{var\_}\mathtt{l}'$ represent the type of var *before* and *after* the update, where $\mathtt{l}'$ is fresh. Constraint $[\![\mathsf{var\_}\mathtt{l}]\!] \ \leq \ [\mathsf{m} : ([\![\mathsf{var\_}\mathtt{l}.\mathsf{m}]\!], \circ)]$ requires var to have member m with annotation at least $\circ$ before the update, while $[\![\mathsf{var\_}\mathtt{l}']\!] \ \leq$ $[\mathsf{m} : ([\![\mathsf{var\_}\mathtt{l}'.\mathsf{m}]\!], \bullet)]$ requires var to have member m with annotation definite after the update[6]. The constraint $[\![\mathsf{var\_}\mathtt{l}']\!] \ \lhd_{\mathsf{m}} [\![\mathsf{var\_}\mathtt{l}]\!]$ requires that only member m is affected by the assignment. The remaining constraints require that the type of member m, $[\![\mathsf{var\_}\mathtt{l}'.\mathsf{m}]\!]$, and the overall expression have the type of the right hand side of the assignment. For example, `this.add = addFn` in a pre-environment $\gamma_2 = \{\mathtt{this} : 1, \mathtt{lab} : \mathtt{L}, ...\}$ where $2 \notin \mathtt{L}$, generates the constraints: $[\![\mathtt{this\_1}]\!] \ \leq$ $[\mathtt{add} : ([\![\mathtt{this\_1.add}]\!], \circ)], [\![\mathtt{this\_2}]\!] \ \leq \ [\mathtt{add} : ([\![\mathtt{this\_2.add}]\!], \bullet)], [\![\mathtt{this\_2}]\!] \ \lhd_{\mathtt{add}}$ $[\![\mathtt{this\_1}]\!], [\![\mathtt{addFn}]\!] \ \leq \ [\![\mathtt{this\_2.add}]\!], [\![\mathtt{addFn}]\!] \ \leq \ [\![\mathtt{this\_2.add} = \mathtt{addFn}]\!]$ and the post-environment $\gamma_2[\mathtt{this} \mapsto 2, \mathtt{lab} \mapsto \mathtt{L} \cup \{2\}]$.

In (*new*) a function is used to create an object. The constraint $[\![\mathtt{this\_f}]\!]^{\circ}$ requires the initial `this` for f to have no definite members. The constraint $[\![\mathsf{e}]\!] \ \leq \ [\![\mathsf{x\_f}]\!]$ requires the actual parameter to have a subtype of the formal parameter, where x_f is the type of the formal parameter at the beginning of the function body. The constraint $[\![\mathtt{ret\_f}]\!] \ \leq \ [\![\mathtt{new}\ \mathsf{f(e)}]\!]$ requires the return type of the function to be a subtype of the overall type of the `new` expression . For example, `new Date(1000)` generates constraints: $[\![\mathtt{this\_Date}]\!]^{\circ}, [\![1000]\!] \ \leq \ [\![\mathtt{x\_Date}]\!], [\![\mathtt{ret\_Date}]\!] \ \leq \ [\![\mathtt{newDate(1000)}]\!]$ The rule for global function (*funcCall*) is similar in structure to that for (*new*).

For member access, (*memAcc*), and for assignment where the receiver is not `this` or `x`, (*assignUpd*), the receiver must have the definite member. For example, `x.mSec`, in a $\gamma_1 = \{\mathtt{x} : 2, ...\}$, generates constraint: $[\![\mathtt{x\_2}]\!] \ \leq$ $[\mathtt{mSec} : ([\![\mathtt{x\_2.mSec}]\!], \bullet)]$

For method call, (*methCall*), we consider the label characterizing the occurrence of the call. For a call with label `l` we require the receiver to have a definite member, m, with function type $[\![\mathtt{call\_this\_l}]\!] \times [\![\mathtt{call\_x\_l}]\!] \rightarrow [\![\mathtt{call\_ret\_l}]\!]$, as expressed through the con-

[6] Using $[\![\mathsf{var\_}\mathtt{l}]\!] \ \leq \ [\mathsf{m} : ([\![\mathsf{e}]\!], \_)]$ instead of $[\![\mathsf{var\_}\mathtt{l}]\!] \ \leq \ [\mathsf{m} : ([\![\mathsf{var\_}\mathtt{l}.\mathsf{m}]\!], \_))]$ would have been too restrictive. Namely, a solution would require the type of m to be the same as the type of $[\![\mathsf{e}]\!]$ rather than a supertype.

straint $[\![\mathsf{e}_1]\!] \leq [\mathtt{m} : ([\![\mathtt{call\_this\_l}]\!] \times [\![\mathtt{call\_x\_l}]\!] \to [\![\mathtt{call\_ret\_l}]\!], \bullet)]$[7]. This will ensure that a solution to the constraints will give a type to the member, that is the least upper bound of all the receivers, parameters and return types at the call sites. For example, `x.add(y)` in a pre-environment $\gamma_3 = \{\mathtt{x} : \mathtt{Main}, \mathtt{y} : \mathtt{Main}, \mathtt{lab} : \mathtt{L}, ...\}$ where $5 \notin \mathtt{L}$, generates constraints:

$[\![\mathtt{x\_Main}]\!] \leq [\mathtt{add} : ([\![\mathtt{x\_Main.add}]\!], \bullet)], [\![\mathtt{x\_Main.add}]\!] \leq ([\![\mathtt{call\_this\_5}]\!] \times [\![\mathtt{call\_x\_5}]\!] \to [\![\mathtt{call\_ret\_5}]\!]), [\![\mathtt{x\_Main}]\!] \leq [\![\mathtt{call\_this\_5}]\!], [\![\mathtt{y\_Main}]\!] \leq [\![\mathtt{call\_x\_5}]\!], [\![\mathtt{call\_ret\_5}]\!] \leq [\![\mathtt{x\_Main.add(y\_Main)}]\!]$ and the post-environment $\gamma_3[\mathtt{lab} \mapsto \mathtt{L} \cup \{5\}]$.

For programs, (*Prog*), we collect the constraints generated for each function with a pre-environment mapping `this` and `x` to their respective initial versions and `lab` to the given set of labels.

5.4 Soundness of the Constraints

We now show that the constraints are sound with respect to the type system. Given a solution, S, and pre-environment, γ, we can generate an environment, Γ, as follows: $\Gamma_{\mathsf{gen}}(\gamma, \mathsf{S}) = \{\mathtt{this} \mapsto \mathsf{S}(\mathtt{this\_}\gamma(\mathtt{this})), \mathtt{x} \mapsto \mathsf{S}(\mathtt{x\_}\gamma(\mathtt{x}))\}$.

Theorem 2 guarantees soundness of the constraints at expression level: Given an expression and its constraints, if there is a solution then the type given by the type system is a subtype of that given in the solution. The environments used for type checking are those produced by Γ_{gen} with pre-environments γ and γ'.

Theorem 2. *If* $\gamma \vdash \mathsf{e} : \mathbb{e} \parallel \gamma' \parallel \mathsf{C}$ *and* $\mathsf{S} \vdash \mathsf{C}$ *and* $\Gamma = \Gamma_{\mathsf{gen}}(\gamma, \mathsf{S})$ *and* $\Gamma' = \Gamma_{\mathsf{gen}}(\gamma', \mathsf{S})$ *then* $\mathsf{P}, \Gamma \vdash \mathsf{e} : \mathsf{t} \parallel \Gamma'$ *and* $\mathsf{t} \leq \mathsf{S}([\![\mathbb{e}]\!])$.

Theorem 3 states soundness of the constraints at the program level. Given a program and its constraints, if there is a solution we can use it to generate a well-typed version of the program. Given a JS_0 program and a solution, function $\mathcal{T}(\mathbb{P}, \mathsf{S})$ generates the corresponding typed $\mathrm{JS}_0^{\mathsf{T}}$ program, by using the solution to find the type of the formal parameter, receiver and return type of all the functions and removing the labeling.

Theorem 3. *If* $\vdash \mathsf{P} : \mathsf{C}$ *and* $\mathsf{S} \vdash \mathsf{C}$ *then* $\vdash \mathcal{T}(\mathsf{P}, \mathsf{S})\diamond$

6 From Constraints to Solutions

We now discuss how constraints can be closed to make explicit a solution and how to check that constraints are well-formed. We show how a well-formed set of constraints can be used to generate a solution.

[7] Constraint $[\![\mathsf{e}_1]\!] \leq [\mathtt{m} :([\![\mathsf{e}_1]\!] \times [\![\mathsf{e}_2]\!] \to [\![\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2)]\!], \bullet)]$ would have been too restrictive. Namely, it would require all the receivers of the method to have the same type.

Constraint Generation for Expressions

$$\dfrac{}{\begin{array}{l}\gamma \vdash \mathtt{null}\ :\ \mathtt{null} \parallel \gamma \parallel \emptyset\\ \gamma \vdash \mathtt{n}\ :\ \mathtt{n} \parallel \gamma \parallel \{\llbracket \mathtt{n} \rrbracket\ \leq\ \mathsf{Int}\}\\ \gamma \vdash \mathtt{this}\ :\ \mathtt{this\_}\gamma(\mathtt{this}) \parallel \gamma \parallel \emptyset\\ \gamma \vdash \mathtt{x}\ :\ \mathtt{x\_}\gamma(\mathtt{x}) \parallel \gamma \parallel \emptyset\end{array}}\ (var) \qquad \dfrac{\mathsf{C} = \{\llbracket \mathsf{f} \rrbracket\ \leq\ (\llbracket \mathtt{this\_f} \rrbracket \times \llbracket \mathtt{x\_f} \rrbracket) \rightarrow\ \llbracket \mathtt{ret\_f} \rrbracket\}}{\gamma \vdash \mathsf{f}\ :\ \mathsf{f} \parallel \gamma \parallel \mathsf{C}}\ (funcId)$$

$$\dfrac{\begin{array}{l}\gamma \vdash \mathsf{e}\ :\ \mathbb{e} \parallel \gamma'' \parallel \mathsf{C}'\\ \gamma''(\mathsf{var}) = \mathtt{l}\\ \mathtt{l}' \notin\ \gamma''(\mathtt{lab})\\ \gamma' = \gamma''[\mathsf{var}\ \mapsto\ \mathtt{l}', \mathtt{lab}\ \mapsto (\gamma''(\mathtt{lab})\ \cup\ \{\mathtt{l}'\})\ ]\\ \ \mathsf{C} = \{\llbracket \mathsf{var\_l} \rrbracket\ \leq\ [\mathsf{m} : (\llbracket \mathsf{var\_l.m} \rrbracket, \circ)],\\ \qquad \llbracket \mathsf{var\_l}' \rrbracket\ \leq\ [\mathsf{m} : (\llbracket \mathsf{var\_l}'\mathsf{.m} \rrbracket, \bullet)],\\ \qquad \llbracket \mathsf{var\_l}' \rrbracket\ \lhd_{\mathsf{m}} \llbracket \mathsf{var\_l} \rrbracket,\ \llbracket \mathbb{e} \rrbracket\ \leq\ \llbracket \mathsf{var\_l}'\mathsf{.m} \rrbracket,\ \ \llbracket \mathbb{e} \rrbracket\ \leq\ \llbracket \mathsf{var\_l}'\mathsf{.m} = \mathbb{e} \rrbracket\}\end{array}}{\gamma \vdash \mathsf{var.m} = \mathsf{e}\ :\ \mathsf{var\_l}'\mathsf{.m} = \mathbb{e} \parallel \gamma' \parallel \mathsf{C}\ \cup\ \mathsf{C}'}\ (assignAdd)$$

$$\dfrac{\begin{array}{l}\gamma \vdash \mathsf{e}\ :\ \mathbb{e} \parallel \gamma' \parallel \mathsf{C}'\\ \mathsf{C} = \{\llbracket \mathtt{this\_f} \rrbracket^{\circ},\ \ \llbracket \mathbb{e} \rrbracket\ \leq\ \llbracket \mathtt{x\_f} \rrbracket,\\ \qquad \llbracket \mathtt{ret\_f} \rrbracket\ \leq\ \llbracket \mathtt{new}\ \mathsf{f}(\mathbb{e}) \rrbracket\}\end{array}}{\gamma \vdash \mathtt{new}\ \mathsf{f}(\mathsf{e})\ :\ \mathtt{new}\ \mathsf{f}(\mathbb{e}) \parallel \gamma' \parallel \mathsf{C}\ \cup\ \mathsf{C}'}\ (new) \qquad \dfrac{\begin{array}{l}\gamma \vdash \mathsf{e}\ :\ \mathbb{e} \parallel \gamma' \parallel \mathsf{C}'\\ \mathsf{C} = \{\llbracket \mathtt{this\_f} \rrbracket^{\circ},\ \ \llbracket \mathbb{e} \rrbracket\ \leq\ \llbracket \mathtt{x\_f} \rrbracket,\\ \qquad \llbracket \mathtt{ret\_f} \rrbracket\ \leq\ \llbracket \mathsf{f}(\mathbb{e}) \rrbracket\}\end{array}}{\gamma \vdash \mathsf{f}(\mathsf{e})\ :\ \mathsf{f}(\mathbb{e}) \parallel \gamma' \parallel \mathsf{C}\ \cup\ \mathsf{C}'}\ (funcCall)$$

$$\dfrac{\gamma \vdash \mathsf{e}\ :\ \mathbb{e} \parallel \gamma' \parallel \mathsf{C}'}{\gamma \vdash \mathsf{e.m}\ :\ \mathbb{e}\mathsf{.m} \parallel \gamma' \parallel \mathsf{C}\ \cup\ \{\llbracket \mathbb{e} \rrbracket\ \leq\ [\mathsf{m} : (\llbracket \mathbb{e}\mathsf{.m} \rrbracket, \bullet)]\}}\ (memAcc)$$

$$\dfrac{\begin{array}{l}\gamma \vdash \mathsf{e}_1\ :\ \mathbb{e}_1 \parallel \gamma' \parallel \mathsf{C}'\\ \gamma' \vdash \mathsf{e}_2\ :\ \mathbb{e}_2 \parallel \gamma'' \parallel \mathsf{C}''\\ \mathsf{C} = \{\llbracket \mathbb{e}_1 \rrbracket\ \leq\ [\mathsf{m} : (\llbracket \mathbb{e}_1\mathsf{.m} \rrbracket, \bullet)],\ \ \llbracket \mathbb{e}_2 \rrbracket\ \leq\ \llbracket \mathbb{e}_1\mathsf{.m} \rrbracket,\ \llbracket \mathbb{e}_2 \rrbracket\ \leq\ \llbracket \mathbb{e}_1\mathsf{.m} = \mathbb{e}_2 \rrbracket\}\end{array}}{\gamma \vdash \mathsf{e}_1\mathsf{.m} = \mathsf{e}_2\ :\ \mathbb{e}_1\mathsf{.m} = \mathbb{e}_2 \parallel \gamma'' \parallel \mathsf{C}\ \cup\ \mathsf{C}'\ \cup\ \mathsf{C}''}\ (assignUpd)$$

$$\dfrac{\begin{array}{l}\gamma \vdash \mathsf{e}_1\ :\ \mathbb{e}_1 \parallel \gamma' \parallel \mathsf{C}'\\ \gamma' \vdash \mathsf{e}_2\ :\ \mathbb{e}_2 \parallel \gamma'' \parallel \mathsf{C}''\\ \mathtt{l}\ \notin\ \gamma''(\mathtt{lab})\\ \mathsf{C} = \{\llbracket \mathbb{e}_1 \rrbracket\ \leq\ [\mathsf{m} : (\llbracket \mathbb{e}_1\mathsf{.m} \rrbracket, \bullet)],\ \llbracket \mathbb{e}_1\mathsf{.m} \rrbracket\ \leq\ ((\llbracket \mathtt{call\_this\_l} \rrbracket \times \llbracket \mathtt{call\_x\_l} \rrbracket) \rightarrow\ \llbracket \mathtt{call\_ret\_l} \rrbracket),\\ \qquad \llbracket \mathbb{e}_1 \rrbracket\ \leq\ \llbracket \mathtt{call\_this\_l} \rrbracket,\ \llbracket \mathbb{e}_2 \rrbracket\ \leq\ \llbracket \mathtt{call\_x\_l} \rrbracket,\ \llbracket \mathtt{call\_ret\_l} \rrbracket\ \leq\ \llbracket \mathbb{e}_1\mathsf{.m}(\mathbb{e}_2) \rrbracket\}\end{array}}{\gamma \vdash \mathsf{e}_1\mathsf{.m}(\mathsf{e}_2)\ :\ \mathbb{e}_1\mathsf{.m}(\mathbb{e}_2) \parallel \gamma''[\mathtt{lab}\ \mapsto (\gamma''(\mathtt{lab})\ \cup\ \{\mathtt{l}\}] \parallel \mathsf{C}\ \cup\ \mathsf{C}'\ \cup\ \mathsf{C}''}\ (methCall)$$

$$\dfrac{\begin{array}{l}\gamma \vdash \mathsf{e}_1\ :\ \mathbb{e}_1 \parallel \gamma' \parallel \mathsf{C}'\\ \gamma' \vdash \mathsf{e}_2\ :\ \mathbb{e}_2 \parallel \gamma'' \parallel \mathsf{C}''\\ \mathsf{C} = \{\llbracket \mathbb{e}_2 \rrbracket\ \leq\ \llbracket \mathbb{e}_1; \mathbb{e}_2 \rrbracket\}\end{array}}{\gamma \vdash \mathsf{e}_1; \mathsf{e}_2\ :\ \mathbb{e}_1; \mathbb{e}_2 \parallel \gamma'' \parallel \mathsf{C}\ \cup\ \mathsf{C}'\ \cup\ \mathsf{C}''}\ (seq) \qquad \dfrac{\begin{array}{l}\gamma \vdash \mathsf{e}\ :\ \mathbb{e} \parallel \gamma' \parallel \mathsf{C}'\\ \gamma'(\mathtt{x}) = \mathtt{l}\\ \mathsf{C} = \{\llbracket \mathbb{e} \rrbracket\ \leq\ \llbracket \mathtt{x\_l} \rrbracket,\ \llbracket \mathbb{e} \rrbracket\ \leq\ \llbracket \mathtt{x\_l} = \mathbb{e} \rrbracket\}\end{array}}{\gamma \vdash \mathtt{x}\ = \mathsf{e}\ :\ \mathtt{x\_l} = \mathbb{e} \parallel \gamma' \parallel \mathsf{C}\ \cup\ \mathsf{C}'}\ (varAss)$$

Constraint Generation for Programs

$$\dfrac{\begin{array}{l}\mathsf{P} = \mathtt{function}\ \mathsf{f}_1(\mathtt{x})\ \{\ \mathsf{e}_1\ \}\ \cdots\ \mathtt{function}\ \mathsf{f}_n(\mathtt{x})\ \{\ \mathsf{e}_\mathsf{n}\ \}\\ \{\mathtt{this}\ \mapsto\ \mathsf{f}_i,\ \mathtt{x}\ \mapsto\ \mathsf{f}_i,\ \mathtt{lab}\ \mapsto\ \gamma'_{i-1}(\mathtt{lab})\} \vdash \mathsf{e}_\mathsf{i}\ :\ \mathbb{e}_i \parallel \gamma'_i \parallel \mathsf{C}_i \quad 1 \leq i \leq n\ \wedge\ \gamma_0 = \emptyset\\ \mathsf{C} = \bigcup_{i\ \in\ 1..n} \mathsf{C}_i \cup\ \{\llbracket \mathbb{e}_\mathsf{i} \rrbracket\ \leq\ \llbracket \mathtt{ret\_f_i} \rrbracket\}\end{array}}{\vdash \mathsf{P} :\ \ \mathsf{C}}\ (Prog)$$

Fig. 11. Constraint Generation

6.1 Constraint Closure

To simplify the extraction of a solution from a set of constraints we apply constraint closure, which makes the solutions (or lack of) explicit. The closing relation, $\mathsf{C} \longrightarrow \mathsf{C}'$, is defined in Figure 12[8].

In (*closeTrans*) we add a constraint implied by the transitivity of subtyping.

In (*closeTransMem*) the type variable τ is required to have the same members as τ' with the same types definite annotation for m (because of $\tau \lhd_{\mathsf{m}} \tau'$); the type variable τ' is required to have member m with type τ'' and annotation ψ (because of $\tau' \leq [\mathsf{m} : (\tau'', \psi)]$). Therefore, τ is also required to have the member m with type τ'' and annotation ψ, as expressed by $\tau \leq [\mathsf{m} : (\tau'', \psi)]$.

In (*closeBalance*) the type variable τ is required to be a subtype of τ', and τ is required to be a subtype of a σ, *i.e.* either of Int, or of a function type. Because the subtype relationship for Int and function types is the identity, it follows that τ and σ will have to be "the same", and therefore, it follows that τ' will have to be a subtype of σ.

In (*closeBalanceMem*) the type variable τ is required to have member m with type τ'' and annotation ψ. Because τ' is required to have the same members as τ with the same types it follows that τ' will also have the member m with type τ''. The annotation, ψ, depends on whether $\mathsf{m} = \mathsf{m}'$. If $\mathsf{m} = \mathsf{m}'$ then ψ can be less defined i.e. $\circ$ otherwise the annotations for m in τ and τ' must be the same.

In (*closeCong*) the same type variable, τ, is required to contain a member m with type τ' and also with type τ''. It follows that τ' should be "equivalent" with τ''. Similarly, in (*closeCongFunc*) because τ is required to be a subtype of two function types, it follows that the two function types should be "equivalent", which, because of the subtype rules for function types, implies that the receiver, argument and return types should be "equivalent".

Thus, for $[\![\texttt{this\_2}]\!] \lhd_{\mathsf{add}} [\![\texttt{this\_1}]\!]$ and $[\![\texttt{this\_1}]\!] \leq [\texttt{mSec} : ([\![\texttt{this\_1.mSec}]\!], \bullet)]$ application of (*closeTransMem*) generates $[\![\texttt{this\_2}]\!] \leq [\texttt{mSec} : ([\![\texttt{this\_1.mSec}]\!], \bullet)]$, which ensures that $[\![\texttt{this\_2}]\!]$ will have member $\texttt{mSec}$. Also, closing $[\![\texttt{this\_2}]\!] \lhd_{\mathsf{add}} [\![\texttt{this\_1}]\!]$ and $[\![\texttt{this\_2}]\!] \leq [\mathsf{add} : ([\![\texttt{this\_2.add}]\!], \bullet)]$ with (*closeBalanceMem*) generates $[\![\texttt{this\_1}]\!] \leq [\mathsf{add} : ([\![\texttt{this\_2.add}]\!], \circ)]$. Lastly, $[\![\texttt{this\_2}]\!] \leq [\mathsf{add} : ([\![\texttt{this\_1.add}]\!], \circ)]$ and $[\![\texttt{this\_2}]\!] \leq [\mathsf{add} : ([\![\texttt{this\_2.add}]\!], \bullet)]$, closed with rule (*closeCong*) generate $[\![\texttt{this\_1.add}]\!] \leq [\![\texttt{this\_2.add}]\!]$, and $[\![\texttt{this\_2.add}]\!] \leq [\![\texttt{this\_1.add}]\!]$.

Definition 4. C *is closed,* $\vdash \mathsf{C} \diamond_{cl}$*, if for any* C'*:* $\mathsf{C} \longrightarrow \mathsf{C}'$ *implies that* $\mathsf{C} = \mathsf{C}'$.

Lemma 2 states that a set of constraints and its closure have the same set of solutions.

Lemma 2. *If* $\mathsf{S} \vdash \mathsf{C}$ *and* $\mathsf{C} \longrightarrow \mathsf{C}'$ *then* $\mathsf{S} \vdash \mathsf{C}'$.

[8] We assume the closure of a set of constraints includes the reflexive closure.

$$\frac{\begin{array}{l}\mathsf{c}_1,...,\mathsf{c}_n \longrightarrow \mathsf{c}'_1,...\mathsf{c}'_m\\ \mathsf{c}_1,...,\mathsf{c}_n \in \mathsf{C}\end{array}}{\mathsf{C} \longrightarrow \mathsf{C} \cup \{\mathsf{c}'_1,...\mathsf{c}'_m\}}\,(closeMany1) \qquad \frac{}{\mathsf{C} \longrightarrow \mathsf{C}}\,(closeMany2)$$

$$\frac{}{\tau \le \tau',\ \tau' \le \rho \longrightarrow \tau \le \rho}\,(closeTrans)$$

$$\frac{}{\tau \lhd_{\_} \tau',\ \tau' \le [\mathsf{m} : (\tau'', \psi)] \longrightarrow \tau \le [\mathsf{m} : (\tau'', \psi)]}\,(closeTransMem)$$

$$\frac{}{\tau \le \tau',\ \tau \le \sigma \longrightarrow \tau' \le \sigma}\,(closeBalance)$$

$$\frac{\psi' = \circ\ (if\ \mathsf{m} = \mathsf{m}')\quad \psi' = \psi\ (otherwise)}{\tau \lhd_{\mathsf{m}'} \tau',\ \tau \le [\mathsf{m} : (\tau'', \psi)] \longrightarrow \tau' \le [\mathsf{m} : (\tau'', \psi')]}\,(closeBalanceMem)$$

$$\frac{}{\tau \le [\mathsf{m} : (\tau', \_)],\ \tau \le [\mathsf{m} : (\tau'', \_)] \longrightarrow \tau' \le \tau'',\ \tau'' \le \tau'}\,(closeCong)$$

$$\frac{}{\begin{array}{l}\tau \le (\tau_1 \times \tau_2 \to \tau_3),\ \tau \le (\tau'_1 \times \tau'_2 \to \tau'_3) \longrightarrow\\ \qquad\qquad \tau'_1 \le \tau_1,\ \tau_1 \le \tau'_1,\ \tau'_2 \le \tau_2,\ \tau_2 \le \tau'_2,\ \tau_3 \le \tau'_3,\ \tau'_3 \le \tau_3\end{array}}\,(closeCongFunc)$$

Fig. 12. Constraint Closure

6.2 Well-Formed Constraints

The well-formedness of constraints, $\vdash \mathsf{C}\diamond$ (shown in Figure 13), ensures that a set of constraints can be used to create a solution. For a set of constraints, C, to be well-formed they must be closed and all the constraints in C must be well-formed. We define function $\mathcal{A}(\mathsf{C}, \tau, \mathsf{m})$ which determines the annotations that any solution satisfying C should give to m in τ. This is done by looking for constraints detailing members: $\tau \le [\mathsf{m} : (\_, \_)]$. A member is annotated with $\circ$ if there are no constraints indicating it should be definite:

$$\mathcal{A}(\mathsf{C}, \tau, \mathsf{m}) = \begin{cases} \bullet & \text{if } \tau \le [\mathsf{m} : (\_ , \bullet)] \in \mathsf{C} \\ \circ & \text{if } \tau \le [\mathsf{m} : (\_ , \circ)] \in \mathsf{C} \text{ and } \tau \le [\mathsf{m} : (\_ , \bullet)] \notin \mathsf{C} \\ \mathcal{U}df & \text{otherwise} \end{cases}$$

Intuitively, rule $(wlfNoDefs)$ corresponds to the solution satisfaction rule $(solNoDefs)$ in Figure 10. Where $\mathsf{S}\ (\tau)(\mathsf{m})$ is represented by looking for constraints detailing members, $\tau \leq [\mathsf{m} : (\_\,,\_)]$, with $\mathcal{A}(\mathsf{C}, \tau, \mathsf{m})$ being used to find the appropriate annotation.

Rules $(wlfMix1)$, $(wlfMix2)$ and $(wlfMix3)$ ensure that the constraints cannot mix object types with function types or integers.

$$\frac{\begin{array}{l}\vdash \mathsf{C} \diamond_{cl} \\ \mathsf{C} = \{\mathsf{c}_1 ... \mathsf{c}_n\} \\ \mathsf{C} \vdash \mathsf{c}_i \quad \forall\, i \in 1...n\end{array}}{\vdash \mathsf{C} \diamond} \,(wlfAll)$$

$$\frac{\tau \leq [\mathsf{m} : (\_\,, \bullet)] \notin \mathsf{C}}{\mathsf{C} \vdash \tau^{\circ}} \,(wlfNoDefs) \qquad \frac{\tau \leq (\_ \times \_ \to \_) \notin \mathsf{C} \,\wedge\, \tau \leq \mathsf{Int} \notin \mathsf{C}}{\mathsf{C} \vdash \tau \leq [\mathsf{m} : (\tau', \psi)]} \,(wlfMix1)$$

$$\frac{\tau \leq [\mathsf{m} : (\_\,, \_\,)] \notin \mathsf{C} \,\wedge\, \tau \leq \mathsf{Int} \notin \mathsf{C}}{\mathsf{C} \vdash \tau \leq (\tau_1 \times \tau_2 \to \tau_3)} \,(wlfMix2)$$

$$\frac{\tau \leq (\_ \times \_ \to \_) \notin \mathsf{C} \,\wedge\, \tau \leq [\mathsf{m} : (\_\,, \_\,)] \notin \mathsf{C}}{\mathsf{C} \vdash \tau \leq \mathsf{Int}} \,(wlfMix3)$$

Fig. 13. Well-formed Constraints

6.3 From Constraints to Solutions

We now show how well-formed constraints, $\vdash \mathsf{C} \diamond$, can be translated into a solution. We first define a *type variable function*, V, from type variables to variables in the type system, $\alpha_1 ... \alpha_n \in ObjVar$. We say that V is well-formed for C, i.e. $\mathsf{C} \vdash \mathsf{V}\diamond$, iff $\tau \leq \tau', \tau' \leq \tau \in \mathsf{C}$ and $\mathsf{V}(\tau) = \alpha$ implies $\mathsf{V}(\tau') = \alpha$.

The translation relation, $\mathsf{C}, \mathsf{V}, \tau \to \mathsf{tp}, \mathsf{V}'$, in Figure 14 translates a type variable, τ, into a type. If a type variable has no constraints indicating whether it should be an object, function or integer type, we default to making it an object type with no members. The extension of V, which is denoted by $\mathsf{V} \oplus \tau$, is defined as follows (where α is a fresh variable):

$$(\mathsf{V} \oplus \tau)(\tau') = \begin{cases} \alpha & \text{if } \tau' \notin dom(\mathsf{V}) \text{ and} \\ & \quad (\ \tau' = \tau \text{ or } (\ \tau' \leq \tau \in \mathsf{C} \text{ and } \tau \leq \tau' \in \mathsf{C}\ )\ ) \\ \mathsf{V}(\tau') & \text{if } \tau' \in dom(\mathsf{V}) \\ \mathcal{U}df & \text{otherwise} \end{cases}$$

$\mathsf{V} \oplus \tau$ extends V with new type variables thus, keeping track of type variables that have already been translated. Because each step of the translation either extends V or finishes when $\mathsf{V}(\tau) = \alpha$ (or $\mathsf{tp} = \mathsf{Int}$ or $\mathsf{tp} = \mu\ \alpha.[\,]$) termination is guaranteed.

$$\frac{\mathsf{V}(\tau) = \alpha}{\mathsf{C}, \mathsf{V}, \tau \to \alpha, \mathsf{V}} \qquad \frac{\tau \ \le\ \mathsf{Int} \in \mathsf{C}}{\mathsf{C}, \mathsf{V}, \tau \to \mathsf{Int}, \mathsf{V}} \qquad \frac{\begin{array}{l} \tau \ \le\ [\mathsf{m} : \_\,] \notin \mathsf{C} \\ \tau \ \le\ (\_ \times \_ \to \_) \notin \mathsf{C} \\ \tau \ \le\ \mathsf{Int} \notin \mathsf{C} \\ \mathsf{V}' = \mathsf{V} \oplus \tau \\ \mathsf{V}'(\tau) = \alpha \end{array}}{\mathsf{C}, \mathsf{V}, \tau \to \mu\ \alpha.[\,], \mathsf{V}'}$$

$$\frac{\begin{array}{l} n \ge 1 \\ \mathsf{V}(\tau) = \mathcal{U}df \\ \mathsf{V}_0 = \mathsf{V} \oplus \tau \\ \mathsf{V}_0(\tau) = \alpha \\ \{\mathsf{m}_1 \ldots \mathsf{m}_n\} = \{\mathsf{m} \mid \tau \ \le\ [\mathsf{m} : (\_, \_\,)] \ \in\ \mathsf{C}\ \} \\ \tau \ \le\ [\mathsf{m}_i : (\tau_i, \_\,)] \ \in\ \mathsf{C}\ (\mathit{for}\ i\ \in 1 \ldots n) \\ \mathsf{C}, \mathsf{V}_{i-1}, \tau_i \to \mathsf{tp}_i, \mathsf{V}_i \\ \psi_i = \mathcal{A}(\mathsf{C}, \tau, \mathsf{m}_i) \end{array}}{\mathsf{C}, \mathsf{V}, \tau \to \mu\ \alpha.[\mathsf{m}_1 : (\mathsf{tp}_1, \psi_1) \ldots \mathsf{m}_n : (\mathsf{tp}_n, \psi_n)], \mathsf{V}_n} \qquad \frac{\begin{array}{l} \mathsf{V}(\tau) = \mathcal{U}df \\ \mathsf{V}_0 = \mathsf{V} \oplus \tau \\ \mathsf{V}_0(\tau) = \alpha \\ \tau \ \le\ (\tau_1 \times \tau_2 \to \tau_3) \in \mathsf{C} \\ \mathsf{C}, \mathsf{V}_{i-1}, \tau_i \to \mathsf{tp}_i, \mathsf{V}_i \quad (\mathit{for}\ i\ \in 1 \ldots 3) \end{array}}{\mathsf{C}, \mathsf{V}, \tau \to \mu\ \alpha.(\mathsf{tp}_1 \times \mathsf{tp}_2 \to \mathsf{tp}_3), \mathsf{V}_3}$$

Fig. 14. Generating the Solution

6.4 Main Result

Lemma 3 states that if two type variables are "equivalent", $\tau \ \le\ \tau', \tau' \ \le\ \tau \in \mathsf{C}$, by a set of well-formed constraints, they will translate to congruent types or the same variable.

Lemma 3. *If* $\vdash \mathsf{C}\diamond$ *and* $\mathsf{C} \vdash \mathsf{V}\diamond$ *and* $\mathsf{C}, \mathsf{V}, \tau \to \mathsf{tp}, \mathsf{V}'$ *and* $\mathsf{C}, \mathsf{V}, \tau' \to \mathsf{tp}', \mathsf{V}''$ *and* $\tau \ \le\ \tau', \tau' \ \le\ \tau \in \mathsf{C}$ *then* $\mathsf{C} \vdash \mathsf{V}'\diamond$ *and* $\mathsf{C} \vdash \mathsf{V}''\diamond$ *and* ($\mathsf{tp} \equiv \mathsf{tp}'$ *or* $\exists\alpha : \mathsf{tp} = \alpha = \mathsf{tp}'$).

Given a well-formed set of constraints and well-formed type variable function we define a generated solution, $\mathsf{S}_{\mathsf{C},\mathsf{V}}$, such that $\mathsf{S}_{\mathsf{C},\mathsf{V}}(\tau) = \mathsf{t}$ if and only if $\mathsf{C}, \mathsf{V}, \tau \to \mathsf{t}, \mathsf{V}'$. Theorem 4 states that a generated solution from a well-formed set of constraints is well-formed.

Theorem 4. *If* $\vdash \mathsf{C}\diamond$ *then* $\mathsf{S}_{\mathsf{C},\emptyset} \vdash \mathsf{C}$.

7 Related Work

Recursive Types and Subtyping Our choice of a recursive types was motivated by the need to allow typing of a large number of JavaScript programs, but at the same time make possible the development of a type inference algorithm. Hence, we have not considered more expressive type systems such as [19].

Type systems for object based languages have been developed mainly in a functional setting, see [1] and [14]. In [22] a type system is defined for the Abadi Cardelli object calculus with concatenation that uses recursive types. The definition of the object types is like ours (without function types) with width subtyping.

Subtyping for recursive function types (that are a subset of our types) has been considered in [4] where subtyping is contravariant on the input types and covariant on the return type. In our paper we have adopted congruence for subtyping between function types, because our aim is not to study the interaction between subtyping and recursive type (as in [4]) but to have a type system allowing type inference.

An imperative, type safe object oriented language, TOIL, was introduced in [8]. Even though the language is class based, its type system does not identify types with classes. This makes the definition of types similar to ours. TOIL, however, does not have extensible objects, so there is no need for identifying potential members.

Dynamic Addition of Members Extensible objects are considered in a functional setting in [13]. An imperative calculus for extensible objects was proposed by Bono and Fisher, in [7]. In their type system there are two types for objects: the *proto*-types that can be extended and the *object*-types that cannot. The type system tracks potential members. The main difference between our type system and their's is that we use recursive types (instead of row types plus universal and existential quantification). This makes it possible to have a decidable type inference algorithm. Note that, Bono and Fisher's aim was to encode classes in their object calculus, not to obtain a type inference algorithm.

In [24] Thiemann gives a type system for a considerable subset of JavaScript. Types are based on discriminative sums with two levels. The outer level determines what kind of base type e.g. number, string, object etc. The inner level determines the features of the type such as the value e.g. the singleton type `Number(100)`. Row types are used to detail the members of an object type. The type system models the automatic conversions that occur in JavaScript through a matching relation. As all conversions are tracked it is possible to flag those which could result in dangerous or unexpected behaviour. Access to a non-existent member does not result in a type error. There are no recursive types and no type inference algorithm is given but there is an implementation.

In the context of type assembly language Morrisett et al. in [17] uses an initialisation flag on the members of type to indicate if they have been assigned to. One could think of the potential and definite annotations of our types as representing the state of initialisation of a member.

Alias types are used in [5] and [9] to track the evolution of objects. In particular, in [9] potential members are used for the same purpose as the current paper. Alias types are, however, very different from the types used in this paper. They are singleton types identified with the address of objects.

Type Inference In [18,21] Palsberg et al. develop a type inference algorithm for a class based language based on flow analysis. The set of types is the class names defined in the program. Each expression, e, is given a type variable, $[\![\mathsf{e}]\!]$, that expresses the - yet unknown - type. They employed a novel approach to model late binding through *conditional constraints*. A conditional constraint has the form $\mathsf{t} \in [\![\mathsf{e}]\!] \implies \mathsf{C}$ saying that constraints C are only applicable when t is a possible type for $[\![\mathsf{e}]\!]$. In [3] this work is applied to the object based language SELF[12]. Each occurrence of an object is given a unique token ω. Thus, object structure is derived from the program. Our work differs in that we must infer the structure of objects. With StarKiller[23] Salib uses the Cartesian Product Algorithm[2] to infer types for Python programs in order to improve compiled code. Object types maintain a reference to their definition when a member is added or updated new object types are generated and propagated through the system.

In [20] Palsberg considers type inference for the first order type system (with recursive types and subtyping) for the Abadi Cardelli object calculus[1]. The system of constraints is a subset of those used in this paper, with two kinds $\tau \leq \tau'$ and $\tau \leq [\mathsf{m} : (\tau, \psi)]$. Furthermore, the Abadi Cardelli calculus does not allow member addition like JS_0. The type system uses a subsumption rule which is encoded in the system by having two type variables for each program point, one before subtyping and one after. For variables there is x and $[\![\mathsf{x}]\!]$ and member access, $[\![\mathsf{e.m}]\!]$ and $< \mathsf{e.m} >$. Instead of a subsumption rule, our type system uses the subtype relation where necessary *e.g.* the actual parameter being a subtype of the formal parameter. Hence, the subtype relation is always used *explicitly* between the types of expressions in the program. Therefore, we don't need to use two type variables to model the application of subsumption. After the constraints are generated a graph is generated and closed. A well-formedness criteria is given to graphs which are then converted to an automata which is used to annotate the program. Our work differs in that we specify closure and well-formedness in terms of constraints rather than convert to a graph.

In [10] Eifrig et al. consider type inference for the class-based language *I-LOOP*. The types are *recursively constrained* in that a type is supplemented with a set of constraints, $\tau\backslash\mathsf{C}$. They take a different approach to us by defining type rules that generate constraints and then modifying the rules to make a deterministic and complete inference system. Fields and methods of a type are detailed with constraints of the form $\tau \leq \textbf{Inst}\ \mathsf{m} : \tau'$, which states that τ has a field m of type τ'.

In [25], Wang et al. give a type inference system for Java that can statically verify the correctness of downcast. The types used are based on those used in [10] as described above. There are types that describe the structure of objects, **obj** $(\delta, [\overline{\mathtt{l}_i : \tau_i}])$ where δ and $\mathtt{l}_i$ are abstract labels for the class name and fields/methods respectively. The structure of the object types is derived from the class structure. Unlike our treatment of method call sites, where we *always* allocate new type variables, they delegates this to closure. By parameterizing closure with a mapping it is possible to *share* type variables between different invocations of a method.

8 Conclusions and Further Work

In this paper a flexible type system for an idealized version of JavaScript is presented, its soundness is outlined, and a type inference algorithm for this type system is defined. The type inference algorithm is sound with respect to the type system. We show how well-formed constraints can be used to generate a solution and annotate an untyped JS_0 program. The main challenges for both the type system and the inference are the imperative nature of the language combined with the possibility of extending objects.

For future work we want to study the completeness of the type inference algorithm, its complexity, and extend the type system to allow more typeable expressions, e.g., allowing a more flexible subtyping for functions. To show completeness we need principality of the type produced, this is quite difficult to achieve for recursive type systems. We would also like to develop a *mixed mode* system where some of the type annotations are already given by the user. For example, we could provide a typing of the Document Object Model[11] and check code in web pages against it.

Acknowledgements

We would like to thank Mario Coppo, Mariangiola Dezani, Matthew Smith and Alex Buckley for their help and insight and our colleagues at Imperial College Department of Computing and Dipartimento di Informatica of Torino University. We would also like to thank the anonymous ECOOP reviewers.

References

1. Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New York, NY, 1996.
2. Ole Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. In *ECOOP*, 2-26, 1995.
3. Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. Type inference of SELF: Analysis of objects with dynamic and multiple inheritance. *Softw., Pract. Exper.*, 25(9):975–995, 1995.
4. Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.
5. C. Anderson, F. Barbanera, M. Dezani-Ciancaglini, and S. Drossopoulou. Can addresses be types? (a case study: objects with delegation). In *WOOD '03*, volume 82 of *ENTCS*. Elsevier, 2003.

6. Christopher Anderson and Paola Giannini. Type checking for javascript. In *WOOD '04*, volume WOOD of *ENTCS*. Elsevier, 2004. `http://www.binarylord.com/work/js0wood.pdf`.
7. V. Bono and K. Fisher. An Imperative, First-Order Calculus with Object Extension. In *Proc. of ECOOP'98*, volume 1445 of *LNCS*, pages 462–497, 1998. A preliminary version already appeared in Proc. of 5th Annual FOOL Workshop.
8. Kim Bruce, A. Schuett, and R. van Gent. Polytoil: A type safe polymorphic object-oriented language. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1995.
9. F. Damiani and P. Giannini. Alias types for environment aware computations. In *WOOD '03*, volume 82 of *ENTCS*. Elsevier, 2003.
10. Jonathan Eifrig, Scott F. Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *Proc. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)'95*, pages 169–184, New York, NY, 1995. ACM Press.
11. Arnaud Le Hors et al. Document Object Model (DOM) Level 3 Core Specification. Technical report, 1998. `http://www.w3.org/TR/2003/CR-DOM-Level-3-Core-20031107`.
12. Ole Agesen et al. The SELF 4.0 Programmer's Reference Manual. `http://research.sun.com/self/`, 1995.
13. K. Fisher. *Type Systems for Object-Oriented Programming Languages*. PhD thesis, Stanford University, 1996. Available as Stanford Computer Science Technical Report number STAN-CS-TR-98-1602.
14. K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994. A preliminary version appeared in *Proc. of IEEE Symp. LICS'93*.
15. David Flanagan. *JavaScript - The Definitive Guide*. O'Reilly, 1998.
16. ECMAScript Language Specification. ECMA International. ECMA-262, 3rd edition, december 1999. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`.
17. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.
18. Nicholas Oxhoj, Jens Palsberg, and Michael I. Schwartzbach. Making type inference practical. In *ECOOP*, pages 329–349, 1992.
19. W. Hill W. Olthoff P. Canning, W. Cook and J. C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proc. Conf. on Functional Programming Languages and Computer Architecture*, pages 273–280. ACM Press, 1989.
20. Jens Palsberg. Efficient inference of object types. *Inf. Comput.*, 123(2):198–209, 1995.
21. Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 26, New York, NY, 1991. ACM Press.
22. Jens Palsberg and Tian Zhao. Type inference for record concatenation and subtyping. *Inf. Comput.*, 189(1):54–86, 2004.
23. Mike Salib. Static Type Inference (for Python) with Starkiller. `http://www.python.org/pycon/dc2004/papers/1/paper.pdf`, 2004.
24. Peter Thiemann. Towards a type system for analyzing javascript programs. In *ESOP*, pages 408–422, 2005.
25. Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for java. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 99–117. Springer-Verlag, 2001.

Chai: Traits for Java-Like Languages

Charles Smith and Sophia Drossopoulou

Department of Computing, Imperial College London

Abstract. Traits support the factoring out of common behaviour, and its integration into classes in a manner that coexists smoothly with inheritance-based structuring mechanisms.

We designed the language *Chai*, which incorporates statically typed traits into a simple Java-inspired base language, and we discuss three versions of the language: *Chai*$_1$, where traits are only a mechanism for the creation of classes; *Chai*$_2$ where traits are a mechanism for the creation of classes, *and* can also introduce types, and *Chai*$_3$ where traits play a role at runtime, and can can be applied to objects, and change the objects' behaviour. We give formal models for these languages, outline the proof of soundness, and our prototype implementation.

1 Introduction

Traits were designed to facilitate code reuse and to assist in structuring large programs. They are conceptually similar to classes, except that they contain no state, only behaviour, and can be combined using a set of simple composition and modification operators. Elements of behaviour that need to be reused in several different parts of a program can be encapsulated in a trait which may be referenced where necessary, avoiding the need to duplicate code.

Traits first appeared in the object-based language Self[22] where they took the form of parent objects to which an object can delegate some of its behaviour. Subsequent work on Traits was based on the class-based language Smalltalk, for which an extension supporting Traits was created[18, 19]. Use of traits can significantly reduce the overall size of libraries[3].

Mixins[5, 4, 11], Multiple Inheritance[21, 14], Family Polymorphism [9], Delegation Layers, and Aspect Oriented Programming share with Traits the aim of code reuse. Traits, like Mixins, and unlike classes in Multiple Inheritance, have no superclasses, and thus are not tied to a particular location in an inheritance hierarchy. Traits and Mixins usually represent composition of incomplete implementations, and thus support decomposition at a finer grain than classes. When a trait is used by a class have the semantics of the class is the same as if the trait methods were part of the class itself — this is called the *flattening* property.

Smalltalk is the first *class based* language on which traits have been applied. While Smalltalk is dynamically typed, our remit was to apply traits to a statically typed, class based language. In this paper we discuss the design and implementation of *Chai*, an extension of a small Java-like language with traits.

We identified three different rôles for traits in *Chai*, and so, we designed three languages:

Chai$_1$ A language like Java, where traits are purely a mechanism to create classes, and where the application of a trait can only be type-checked after the resulting class has been "flattened".

Chai$_2$ Here we extend the remit of traits so that they may be used as types. This permits checking of traits *before* their application.

Chai$_3$ Finally, we allow traits to be substituted for one another dynamically, supporting runtime changes in object behaviour.

In section 2 of this paper we introduce *Chai* through examples. In sections 3, 4 and 5 we present formal models for *Chai*$_1$, *Chai*$_2$ and *Chai*$_3$. In section 6 we discuss a prototype implementation of these languages, and section 7 contains conclusions and future work.

The prototype implementation and the MSc thesis are available at `http://chai-t.sourceforge.net/`. An appendix with complete definitions and handwritten proofs is available at `http://www.doc.ic.ac.uk/~scd/ChaiApp`.

2 An Example

Figure 1 gives an example of *Chai*$_1$.[1] We define four simple traits: `TScreenShape`, `TPrintedShape`, `TEmptyCircle` and `TFilledCircle`, which describe corresponding components in a simple graphics program: we can form on screen, or print, empty circles or filled circles in any combination in which we are interested, simply by creating a subclass of `Circle` that uses the traits providing the behaviour we want. In the example, we show only two of the four combinations, *i.e.,* classes `ScreenEmptyCircle` and `PrintedFilledCircle`.

A trait T may declare *requirements*, *i.e.,* a list of method signatures, for methods that must be provided by classes or traits using T. Here, trait `TEmptyCircle` requires a method `drawPoint` with return type `void`, and two `int` parameters, and method `getradius` with no parameters, and `int` return type. Class `SreenEmptyCircle` uses `TScreenEmptyCircle`; there, the first method is provided by trait `TScreenShape` and the second by class `Circle`.

Forming the four combinations using single inheritance would mean considerable code duplication, although it could be implemented using multiple inheritance or mixins. See [20] for examples of situations where Traits give more elegant solutions than mixins, and also for examples where mixins give more

[1] For the sake of simplicity, *Chai* only allows methods with a single parameter called `x`, and does not permit sequences of expressions. These restrictions have minimal implications for the presentation of the features we are interested in, and are not adhered to by the examples in this section, nor by the prototype implementation.

```
class Circle {
   int radius;
   int getRadius() { ... }
}

trait TEmptyCircle {
   requires {
      void drawPoint(int x,int y);
      int getRadius();
   }
   void draw() { ... }
}

trait TFilledCircle {
   requires {
      void drawPoint(int x,int y);
      int getRadius();
   }
   void draw() { ... }
}
```

```
trait TScreenShape {
   void drawPoint(int x,int y) {
      ...
   }
}

trait TPrintedShape {
   void drawPoint(int x,int y) {
      ...
   }
}

class ScreenEmptyCircle
  extends Circle
  uses TEmptyCircle,TScreenShape { }

class PrintedFilledCircle
  extends Circle
  uses TFilledCircle,TPrintedShape
  { }
```

Fig. 1. *Chai*$_1$ Example

```
class ScreenShapeStack {
   void push(TScreenShape shape) { ... }
   TScreenShape pop() { ... }
   ...
}

ScreenShapeStack stack = new ScreenShapeStack();
stack.push(new ScreenEmptyCircle());
stack.push(new ScreenFilledCircle());
TScreenShape shape = stack.pop();
```

Fig. 2. *Chai*$_2$ Example

elegant solutions than traits. Because the trait composition operators are so flexible, *Chai*$_1$ allows much finer code reuse than Java.

Figure 2 gives an example of the additional features of *Chai*$_2$, where we allow traits to be used as types. Any object of a class using the trait `TScreenShape` can be referenced through a variable of type `TScreenShape`. Allowing traits to define types supports more polymorphism, and it allows us to type check traits independently of the classes that use them.

```
class CircleShape extends Circle uses TEmptyCircle,TScreenShape {}

CircleShape circle = new CircleShape(); circle.draw();
       // draws an empty circle on screen
circle<TEmptyCircle -> TFilledCircle>; circle.draw();
       // draws a filled circle on screen
circle<TScreenShape -> TPrintedShape>; circle.draw();
       //  prints a filled circle
```

Fig. 3. *Chai*$_3$ Example

In the final example, in figure 3, we show how dynamic substitution of traits can change the behaviour of an object at runtime. The object `circle` starts out as an empty circle on screen, but by substitution of `TFilledCircle` for `TEmptyCircle` we can change it to a filled circle, and subsequently by substituting `TPrintedShape` for `TScreenShape` we can change it to become a printed filled circle.

3 The Language *Chai*$_1$

3.1 Syntax

For *Chai*$_1$ we adapted Traits from Smalltalk[18] to the Java setting. As in [18], traits can be used to add behaviour to classes or to other traits. Traits may contain method definitions, but no fields — as we said earlier, traits are *pure behaviour* [18]. A trait cannot itself take part in execution (i.e. it cannot be instantiated) — it (or a trait using it) can be *used* by some class, which can then be instantiated.

Traits are not required to be complete - that is, they may require functionality beyond their own to be provided by classes or traits using them. The requirements are declared explicitly in the form of a set of *required methods*.

A *Chai*$_1$ program consists of trait and class declarations. A trait declaration consists of a name for the trait, an optional list of used traits (whose behaviour it incorporates), and a trait body. The trait body contains method definitions, and "trait glue", *i.e.,* declaration of required methods, exclude declarations (which exclude methods that would otherwise be incorporated from used traits) and alias declarations (which give new labels to methods incorporated from a used trait).

In contrast to the language in [18] which is untyped and where requirements are inferred automatically, in *Chai*$_1$ provided *and required methods* must be declared using a full type signature.[2] In common with [18], we distinguish required methods into those that must be provided by a class using

[2] In a related work [17], a typed language with automatic inference of required methods is described.

program	::==	(*trait* | *class*)*
trait	::==	**trait** *tr* [**uses** *tr*+] { (*trait-glue* | *meth*)* }
field	::==	*type f*;
meth	::==	*meth-sig* { *exp* }
type	::==	*cl*
exp	::==	*exp.f* := *exp* | *exp.m*(*exp*) | **super**.*m*(*exp*) |
		new *cl* | *var* | **this** | **null** | **x**
trait-glue	::==	**requires** { (*meth-sig* ; | *super-sig* ;)* } |
		exclude { (*t.m* ;)* } |
		alias { (*t.m* **as** *m* ;)* }
meth-sig	::==	*type m*(*type* x)
super-sig	::==	*type* **super**.*m*(*type* x)
class	::==	**class** *cl* **extends** *cl* [**uses** *tr*+] { (*field* | *meth*)* }
cl,tr,m,f	::==	*identifiers*

Fig. 4. $Chai_1$ Syntax

the trait, and those that must be provided *by the superclass* of the using class. This was necessary because Java allows explicit access to superclass methods (through the `super.m(...)` construct), and so if a trait `tr` is used by a class `cl`, then superclass method calls inside methods of `tr` will resolve to methods from the superclass of `cl`.

A class declaration consists of a name for the class, its superclass, an optional list of used traits (whose behaviour it incorporates), and a class body. The class body contains method definitions and fields.

For simplicity, our model does not support method overloading or field hiding, however we believe it can be easily extended to do so — the implementation of $Chai_1$ supports it.

Note, that in $Chai_1$ classes form types, but traits do not.

3.2 Basic Lookup Functions

We consider that a program `P` implicitly defines the following eight (partial) lookup functions:

- $\mathtt{P}^{\mathtt{sup}}$(`cl`) returns the direct superclass of `cl` in `P`.
- $\mathtt{P}^{\mathtt{fld}}$(`cl`,`f`) returns the type of field `f` as defined in class `cl`.
- $\mathtt{P}^{\mathtt{mth}}$(`cl`,`m`) and $\mathtt{P}^{\mathtt{mth}}$(`tr`,`m`) return the (possibly empty) set of methods with identifier `m` defined in class `cl` or trait `tr`.
- $\mathtt{P}^{\mathtt{use}}$(`cl`), or $\mathtt{P}^{\mathtt{use}}$(`tr`) return the set of traits directly used in class `cl`, or trait `tr`.
- $\mathtt{P}^{\mathtt{excl}}$(`tr`) returns the set of trait and method identifier pairs excluded from trait `tr`.
- $\mathtt{P}^{\mathtt{alias}}$(`tr`,`m`) returns the set of trait and method identifier pairs which are aliases of method `m` in trait `tr`.

- $\mathtt{P}^{\mathtt{req}}(\mathtt{tr})$ returns the set of method signatures mentioned as required in the declaration of `tr`.
- $\mathtt{P}^{\mathtt{req\text{-}sup}}(\mathtt{tr})$ returns the set of method signatures mentioned as required for the superclass (through `t super.m(t'x)`) in the declaration of `tr`.

Note that the above functions correspond to *direct* lookups in the program text, and do not take class inheritance nor traits use into account. In the next section we will define the functions $\mathcal{F}$, $\mathcal{F}s$, $\mathcal{M}$, $\mathcal{MS}ig$, and $\mathcal{M}^{\mathtt{orig}}$, which lookup fields, and methods, and which *do* take class inheritance and traits use into account.

3.3 Method or Field Acquisition Through Traits Use and Inheritance

The function $\mathcal{F}(\mathtt{P},\mathtt{cl},\mathtt{f})$ looks up the field `f` in `cl` or its superclasses, and returns `t` where `t` is the type of `f` in `cl`. The function $\mathcal{F}_{\mathrm{s}}(\mathtt{P},\mathtt{cl})$ returns the set of fields defined in `cl`, or inherited from `cl`'s superclasses. The two functions operate only on classes (traits have no fields).

$$\begin{aligned}
\mathcal{F}(\mathtt{P},\mathtt{Object},\mathtt{f}) &= \bot\\
\mathcal{F}(\mathtt{P},\mathtt{cl},\mathtt{f}) &= \begin{cases}\mathtt{P}^{\mathtt{fld}}(\mathtt{cl},\mathtt{f}) & \text{if } \mathtt{P}^{\mathtt{fld}}(\mathtt{cl},\mathtt{f}) \neq \bot\\ \mathtt{P}^{\mathtt{fld}}(\mathtt{P}^{\mathtt{sup}}(\mathtt{cl}),\mathtt{f}) & \text{otherwise}\end{cases}\\
\mathcal{F}_{\mathrm{s}}(\mathtt{P},\mathtt{cl}) &= \{\ \mathtt{f}\ \mid \mathcal{F}(\mathtt{P},\mathtt{cl},\mathtt{f}) \neq \bot\ \}
\end{aligned}$$

A class `cl` that uses a trait `tr` acquires the methods from `tr` in such a way that externally there is no way to tell that the methods were not declared by `cl` itself. This forms the basis of the *flattening property* of traits - a trait formed by using existing traits can be viewed as either a composite entity comprising the used traits and the definitions in the new trait, or as a flattened entity containing all the definitions of its constituents.

We now define $\mathcal{M}(\mathtt{P},\mathtt{cl},\mathtt{m})$ and $\mathcal{M}(\mathtt{P},\mathtt{tr},\mathtt{m})$, which recursively search the used traits and superclasses. Intuitively, these functions embody give precedence to "local" declarations: methods defined in a trait body have highest precedence, and methods that have been aliased have higher precedence than methods acquired from used traits. Methods defined in a class body have highest precedence, and methods acquired from used traits have higher precedence than those acquired from superclasses.

$$\mathcal{M} : \mathtt{program} \times (\mathtt{classId} \cup \mathtt{traitId}) \times \mathtt{methodId} \rightarrow \wp(\mathtt{methodBody})$$

If `tr` is a trait :

$$\mathcal{M}(\mathtt{P},\mathtt{tr},\mathtt{m}) = \begin{cases}\mathtt{P}^{\mathtt{mth}}(\mathtt{P}) & \text{if } \mathtt{P}^{\mathtt{mth}}(\mathtt{P}) \neq \emptyset\\ \mathit{MsAlias} & \text{if } \mathtt{P}^{\mathtt{mth}}(\mathtt{P}) = \emptyset \neq \mathit{MsAlias}\\ \mathit{MsUsed} & \text{if } \mathtt{P}^{\mathtt{mth}}(\mathtt{P}) = \emptyset = \mathit{MsAlias}.\end{cases}$$

where

$$\mathit{MsAlias} = \bigcup_{\mathtt{tr}'.\mathtt{m}' \in \mathtt{P}^{\mathtt{alias}}(\mathtt{tr},\mathtt{m})} \mathcal{M}(\mathtt{P},\mathtt{tr}',\mathtt{m}')$$

$$\mathit{MsUsed} = \bigcup_{\mathtt{tr}'' \in \mathtt{P}^{\mathtt{use}}(\mathtt{tr})\ ,\ \mathtt{tr}''.\mathtt{m} \notin \mathtt{P}^{\mathtt{excl}}(\mathtt{tr}.\mathtt{m})} \mathcal{M}(\mathtt{P},\mathtt{tr}'',\mathtt{m})$$

If cl is a class :

$\mathcal{M}(\mathtt{P},\mathtt{Object},\mathtt{m}) = \emptyset$

$$\mathcal{M}(\mathtt{P},\mathtt{cl},\mathtt{m}) \qquad = \begin{cases} \mathtt{P}^{\mathtt{mth}}(\mathtt{cl},\mathtt{m}) & \text{if } \mathtt{P}^{\mathtt{mth}}(\mathtt{cl},\mathtt{m}) \neq \emptyset \\ \mathit{MsUsed} & \text{if } \mathtt{P}^{\mathtt{mth}}(\mathtt{cl},\mathtt{m}) = \emptyset \neq \mathit{MsUsed} \\ \mathcal{M}(\mathtt{P},\mathtt{P}^{\mathtt{sup}}(\mathtt{cl}),\mathtt{m}) & \text{if } \mathtt{P}^{\mathtt{mth}}(\mathtt{cl},\mathtt{m}) = \emptyset = \mathit{MsUsed}, \end{cases}$$

where

$$\mathit{MsUsed} = \bigcup_{\mathtt{tr}\in\mathtt{P}^{\mathtt{use}}(\mathtt{cl})} \mathcal{M}(\mathtt{P},\mathtt{tr},\mathtt{m})$$

Because there are several ways a trait might acquire a method (from any of the traits it uses), the method lookup functions return *sets* of methods. If the precedence rules do not resolve the method lookup to a single method, *i.e.,* if in a class cl, $|\mathcal{M}(\mathtt{P},\mathtt{cl},\mathtt{m})| > 1$ for some m, then a conflict occurs, [3].

A class is *complete* if it has no conflicts, and if any call to super in any inherited method body resolves without conflict:[4]

$$\frac{\forall \mathtt{m}: \quad \mathtt{e} \text{ contains } \mathtt{super.m}(...) \implies \quad |\mathcal{M}(\mathtt{P},\mathtt{P}^{\mathtt{sup}}(\mathtt{cl}),\mathtt{m}')| = 1}{\mathtt{super} \text{ resolves without conflict in } \mathtt{e} \text{ and } \mathtt{cl}}$$

$$\frac{\begin{array}{l}\forall \mathtt{m}\text{: } |\mathcal{M}(\mathtt{P},\mathtt{cl},\mathtt{m})| \leq 1 \\ \forall \mathtt{m},\mathtt{cl}' \quad \mathtt{P} \vdash_{1} \mathtt{cl} \leq \mathtt{cl}',\ ...\{\mathtt{e}\} \in \mathcal{M}(\mathtt{P},\mathtt{cl}',\mathtt{m}) \implies \\ \qquad \mathtt{super} \text{ resolves without conflict in } \mathtt{e} \text{ and } \mathtt{cl}' \end{array}}{\mathtt{P} \vdash_{1} \mathtt{cl} \diamond_{\mathrm{cmpl}}}$$

For class cl, and trait tr, we define the functions $\mathcal{MS}\mathrm{ig}_{1}(\mathtt{P},\mathtt{cl},\mathtt{m})$, $\mathcal{MS}\mathrm{ig}_{1}(\mathtt{P},\mathtt{tr},\mathtt{m})$, and $\mathcal{MS}\mathrm{ig}_{1}^{\mathtt{sup}}(\mathtt{P},\mathtt{cl},\mathtt{m})$ which return the set of signatures for method m as found in cl, tr or the superclass of cl.

$$\begin{array}{ll} \mathcal{MS}\mathrm{ig}_{1}(\mathtt{P},\mathtt{cl},\mathtt{m}) & = \{\ \mathtt{t\ m(t'x)}\ \mid\ \mathtt{t\ m(t'\ x)}\{...\} \in \mathcal{M}(\mathtt{P},\mathtt{cl},\mathtt{m})\ \} \\ \mathcal{MS}\mathrm{ig}_{1}(\mathtt{P},\mathtt{tr},\mathtt{m}) & = \{\ \mathtt{t\ m(t'x)}\ \mid\ \mathtt{t\ m(t'\ x)}\{...\} \in \mathcal{M}(\mathtt{P},\mathtt{tr},\mathtt{m})\ \} \\ \mathcal{MS}\mathrm{ig}_{1}^{\mathtt{sup}}(\mathtt{P},\mathtt{cl},\mathtt{m}) & = \mathcal{MS}\mathrm{ig}_{1}(\mathtt{P},\mathtt{P}^{\mathtt{sup}}(\mathtt{cl}),\mathtt{m}) \end{array}$$

Note, that the look up functions $\mathcal{M}(\mathtt{P},\mathtt{cl},\mathtt{m})$ and $\mathcal{MS}\mathrm{ig}_{1}(\mathtt{P},\mathtt{cl},\mathtt{m})$ abstract from the use of traits. Therefore, as we will see later, we were able to write the operational semantics and type system of *Chai*$_1$ and *Chai*$_2$ without explicit mention of traits.

We define the function $\mathcal{M}^{\mathtt{orig}}$ which determines the "origin" of a method, *i.e.,* the most specific superclass of a class cl which contains a body for m. We will use $\mathcal{M}^{\mathtt{orig}}$ to model the behaviour of super.m(_).[5]

[3] Conflicts can be avoided by overriding the conflicting method in the class where the conflict occurs, or by excluding one of the conflicting methods - in our system without overloading this works only if all conflicting methods have the same signature.

[4] A simpler, but more restrictive, requirement would be to require no conflicts in any of cl's superclasses.

[5] This formalization of super has been suggested to us by Andrew Black and Chuan-Kai Lin, and slightly adapted by Rok Strnisa.

$$
\begin{array}{lll}
\leadsto \; : & \multicolumn{2}{l}{\texttt{program} \;\rightarrow\; \texttt{exp} \times \mathbf{stack} \times \mathbf{heap} \;\rightarrow\; (\mathbf{val} \cup \mathbf{dev}) \times \mathbf{heap}} \\
\mathbf{stack} & = \mathbf{addr} \times \mathbf{val} \times \texttt{classId} & \\
\mathbf{heap} & = \mathbf{addr} \rightarrow \mathbf{object} & \\
\mathbf{val} & = \{ \texttt{null} \} \cup \mathbf{addr} & \\
\mathbf{object} & = \{ \llbracket \texttt{cl} \parallel \texttt{f}_1 : \texttt{v}_1, \ldots, \texttt{f}_\texttt{r} : \texttt{v}_\texttt{r} \rrbracket \mid \texttt{cl} \in \texttt{classId},\ \texttt{f}_1,\ldots,\texttt{f}_\texttt{r} \in \texttt{fldId},\ \texttt{v}_1,\ldots,\texttt{v}_\texttt{r} \in \mathbf{val} \} & \\
\mathbf{addr} & = \{ \iota_n \mid n \text{ is a natural number} \} & \\
\mathbf{dev} & = \{ \texttt{nllPntrExc}, \texttt{stuckExc} \} &
\end{array}
$$

Fig. 5. *Chai*$_1$ Runtime

$$
\mathcal{M}^{\texttt{orig}}(\texttt{P},\texttt{cl},\texttt{m}) = \begin{cases} \texttt{cl} & \text{if } \texttt{P}^{\texttt{mth}}(\texttt{cl},\texttt{m}) \neq \emptyset \quad \text{or} \quad \exists \texttt{tr} \text{ with} \\ & \texttt{tr} \in \texttt{P}^{\texttt{use}}(\texttt{cl}) \text{ and } \mathcal{M}(\texttt{P},\texttt{tr},\texttt{m}) \neq \emptyset, \\ \bot & \text{if } \texttt{cl} = \texttt{Object}, \\ \mathcal{M}^{\texttt{orig}}(\texttt{P},\texttt{cl}',\texttt{m}) & \text{otherwise, for } \texttt{cl}' = \texttt{P}^{\texttt{sup}}(\texttt{cl}). \end{cases}
$$

3.4 Operational Semantics

We give a large step semantics for *Chai*$_1$, where programs map expressions, stacks and heaps, onto results and new heaps. A stack, $\sigma \in \mathbf{stack}$, is a triple consisting of the address of the current receiver, the value of the actual parameter and the class containing the method body currently being executed. The notation $\sigma(\texttt{this})$, $\sigma(\texttt{x})$, and $\sigma(\texttt{this\_class})$ selects the first, second and third component of σ. A heap, $\chi \in \mathbf{heap}$, maps addresses to objects. Objects contain the class of the object (`cl`), and values ($\texttt{v}_\texttt{i}$) for the object's fields ($\texttt{f}_\texttt{i}$).

The operational semantics of *Chai*$_1$ does not mention traits explicitly, and operates entirely in terms of classes; thus it is very similar to that of a small Java-like language (*e.g.,* ClassicJava[11] or *$\mathcal{F}$ickle*[7]), and is rather standard. It is given in figure 6. The receiver and the parameter are looked up in the heap (**var**). If `null` is dereferenced, a `nullPnterExc` exception is thrown (**null-exception**). Field access is evaluated by looking up the particular field in the object (**field**). Field assignment overrides the corresponding field with the value of the right hand side (**field-assign**). Object creation creates a new object of the appropriate class, and initializes all its fields with `null` (**new**). Method call evaluates the method body found in the *dynamic* class of the receiver; evaluation takes place in a stack consisting of the receiver and actual parameter of the call, and the identifier of the class containing the method body (**method-call**). For the call to `super` the evaluation is similar, but the method is looked up in the *static* superclass of the class given by $\sigma(\texttt{this\_class})$, *i.e.,* the superclass of the class containing the method currently being executed (**super-call**). We require the method lookup functions to return a singleton set, *i.e.,* there should be no conflicting method definitions.

For brevity, we omitted the rules throwing `stuckErr` when conflicting methods are called, or non-existent fields or methods are accessed or called, as well as the rules propagating exceptions `stuckExc` or `nullPntrExc`.

null

$$\frac{}{\mathtt{null},\ \sigma,\ \chi \rightsquigarrow_{\mathtt{P}} \mathtt{null}, \chi}$$

null-exception

$$\frac{\mathtt{e},\ \sigma,\ \chi \rightsquigarrow_{\mathtt{P}} \mathtt{null}, \chi'}{\begin{array}{l}\mathtt{e.f} := \mathtt{e}',\ \sigma,\ \chi \rightsquigarrow_{\mathtt{P}} \mathtt{nllPntrExc}, \chi' \\ \mathtt{e.f},\ \sigma,\ \chi \rightsquigarrow_{\mathtt{P}} \mathtt{nllPnterExc}, \chi' \\ \mathtt{e.m}(\mathtt{e}'),\ \sigma,\ \chi \rightsquigarrow_{\mathtt{P}} \mathtt{nllPntrExc}, \chi'\end{array}}$$

field

$$\frac{\mathtt{e},\ \sigma,\ \chi \rightsquigarrow_{\mathtt{P}} \iota, \chi'}{\mathtt{e.f},\ \sigma,\ \chi \rightsquigarrow_{\mathtt{P}} \chi'(\iota)(\mathtt{f}), \chi'}$$

var

$$\frac{}{\begin{array}{l}\mathtt{x},\ \sigma,\ \chi \rightsquigarrow_{\mathtt{P}} \sigma(\mathtt{x}), \chi \\ \mathtt{this},\ \sigma,\ \chi \rightsquigarrow_{\mathtt{P}} \sigma(\mathtt{this}), \chi\end{array}}$$

field-assign

$$\frac{\begin{array}{l}\mathtt{e},\ \sigma,\ \chi \rightsquigarrow_{\mathtt{P}} \iota, \chi'' \\ \mathtt{e}',\ \sigma,\ \chi'' \rightsquigarrow_{\mathtt{P}} \mathtt{v}, \chi''' \\ \chi' = \chi'''[\iota \mapsto \chi'''(\iota)[\mathtt{f} \mapsto \mathtt{v}]]\end{array}}{\mathtt{e.f} := \mathtt{e}',\ \sigma,\ \chi \rightsquigarrow_{\mathtt{P}} \mathtt{v}, \chi'}$$

new

$$\frac{\begin{array}{l}\mathcal{F}_{\mathrm{s}}(\mathtt{P}, \mathtt{cl}) = \mathtt{f}_1, \ldots \mathtt{f}_{\mathtt{r}} \\ \forall \mathtt{k} \in 1, \ldots \mathtt{r} : \mathtt{v}_{\mathtt{k}} = \mathtt{null} \\ \iota \text{ is new in } \chi\end{array}}{\mathtt{new\ cl},\ \sigma,\ \chi \rightsquigarrow_{\mathtt{P}} \iota, \chi[\iota \mapsto [\![\, \mathtt{cl} \,\|\, \mathtt{f}_1 : \mathtt{v}_1, \ldots \mathtt{f}_{\mathtt{r}} : \mathtt{v}_{\mathtt{r}} \,]\!]]}$$

method-call

$$\frac{\begin{array}{l}\mathtt{e}_{\mathtt{r}},\ \sigma,\ \chi \rightsquigarrow_{\mathtt{P}} \iota, \chi_0 \\ \mathtt{e}_{\mathtt{a}},\ \sigma,\ \chi_0 \rightsquigarrow_{\mathtt{P}} \mathtt{v}_1, \chi_1 \\ \chi_1(\iota) = [\![\, \mathtt{cl} \,\|\, \ldots \,]\!] \\ \mathcal{M}(\mathtt{P}, \mathtt{cl}, \mathtt{m}) = \{\ \mathtt{t\ m}(\mathtt{t}'\ \mathtt{x})\ \{\ \mathtt{e}\ \}\ \} \\ \mathcal{M}^{\mathrm{orig}}(\mathtt{P}, \mathtt{cl}, \mathtt{m}) = \mathtt{cl}' \\ \sigma' = (\iota, \mathtt{v}_1, \mathtt{cl}') \\ \mathtt{e},\ \sigma',\ \chi_1 \rightsquigarrow_{\mathtt{P}} \mathtt{v}, \chi'\end{array}}{\mathtt{e}_{\mathtt{r}}.\mathtt{m}(\mathtt{e}_{\mathtt{a}}),\ \sigma,\ \chi \rightsquigarrow_{\mathtt{P}} \mathtt{v}, \chi'}$$

super-call

$$\frac{\begin{array}{l}\mathtt{e}_{\mathtt{a}},\ \sigma,\ \chi \rightsquigarrow_{\mathtt{P}} \mathtt{v}_1, \chi_1 \\ \sigma(\mathtt{this\_class}) = \mathtt{cl} \\ \mathtt{P}^{\mathrm{sup}}(\mathtt{cl}) = \mathtt{cl}'' \\ \mathcal{M}(\mathtt{P}, \mathtt{cl}'', \mathtt{m}) = \{\ \mathtt{t\ m}(\mathtt{t}'\ \mathtt{x})\ \{\ \mathtt{e}\ \}\ \} \\ \mathcal{M}^{\mathrm{orig}}(\mathtt{P}, \mathtt{cl}'', \mathtt{m}) = \mathtt{cl}' \\ \sigma' = (\sigma(\mathtt{this}), \mathtt{v}_1, \mathtt{cl}') \\ \mathtt{e},\ \sigma',\ \chi_1 \rightsquigarrow_{\mathtt{P}} \mathtt{v}, \chi'\end{array}}{\mathtt{super.m}(\mathtt{e}_{\mathtt{a}}),\ \sigma,\ \chi \rightsquigarrow_{\mathtt{P}} \mathtt{v}, \chi'}$$

Fig. 6. *Chai*$_1$ Operational Semantics

$$\frac{\mathtt{P} = \ldots\ \mathtt{class\ cl\ extends\ cl}'\ \ldots}{\begin{array}{l}\mathtt{P} \vdash_1 \mathtt{cl} \leq \mathtt{cl} \\ \mathtt{P} \vdash_1 \mathtt{cl} \leq \mathtt{cl}' \\ \mathtt{P} \vdash \mathtt{cl} \diamond_{\mathrm{class}} \\ \mathtt{P} \vdash_1 \mathtt{cl} \diamond_{\mathrm{type}}\end{array}} \qquad \frac{\begin{array}{l}\mathtt{P} \vdash_1 \mathtt{cl} \leq \mathtt{cl}' \\ \mathtt{P} \vdash_1 \mathtt{cl}' \leq \mathtt{cl}''\end{array}}{\mathtt{P} \vdash_1 \mathtt{cl} \leq \mathtt{cl}''} \qquad \frac{\mathtt{P} = \ldots\ \mathtt{trait\ tr}\ \ldots}{\mathtt{P} \vdash \mathtt{tr} \diamond_{\mathrm{trait}}}$$

Fig. 7. Subclasses and Subtypes in *Chai*$_1$

3.5 Type System

In figure 7 we define the judgements $\mathtt{P} \vdash \mathtt{cl} \leq \mathtt{cl}'$ indicating subtypes, and $\mathtt{P} \vdash \mathtt{cl} \diamond_{\mathrm{class}}$ and $\mathtt{P} \vdash \mathtt{tr} \diamond_{\mathrm{trait}}$ indicating that $\mathtt{cl}$ is a class or $\mathtt{tr}$ is a trait. We also define the judgement $\mathtt{P} \vdash \mathtt{t} \diamond_{\mathrm{type}}$ indicating that $\mathtt{t}$ is a type.

subsumption

$$\frac{\begin{array}{l}\mathtt{P},\Gamma \vdash_1 \mathtt{e} : \mathtt{t} \\ \mathtt{P} \vdash_1 \mathtt{t} \le \mathtt{t}'\end{array}}{\mathtt{P},\Gamma \vdash_1 \mathtt{e} : \mathtt{t}'}$$

var-this

$$\frac{}{\begin{array}{l}\mathtt{P},\Gamma \vdash_1 \mathtt{x} : \Gamma(\mathtt{x}) \\ \mathtt{P},\Gamma \vdash_1 \mathtt{this} : \Gamma(\mathtt{this})\end{array}}$$

new

$$\frac{\mathtt{P} \vdash_1 \mathtt{cl} \diamond_{\mathrm{cmpl}}}{\mathtt{P},\Gamma \vdash_1 \mathtt{new\,cl} : \mathtt{cl}}$$

null

$$\frac{\mathtt{P} \vdash_1 \mathtt{t} \diamond_{\mathrm{type}}}{\mathtt{P},\Gamma \vdash_1 \mathtt{null} : \mathtt{t}}$$

field

$$\frac{\begin{array}{l}\mathtt{P},\Gamma \vdash_1 \mathtt{e} : \mathtt{cl} \\ \mathcal{F}(\mathtt{P},\mathtt{cl},\mathtt{f}) = \mathtt{t}\end{array}}{\mathtt{P},\Gamma \vdash_1 \mathtt{e.f} : \mathtt{t}}$$

field-assign

$$\frac{\begin{array}{l}\mathtt{P},\Gamma \vdash_1 \mathtt{e} : \mathtt{cl} \\ \mathtt{P},\Gamma \vdash_1 \mathtt{e}' : \mathtt{t} \\ \mathcal{F}(\mathtt{P},\mathtt{cl},\mathtt{f}) = \mathtt{t}\end{array}}{\mathtt{P},\Gamma \vdash_1 \mathtt{e.f} := \mathtt{e}' : \mathtt{t}}$$

method-call

$$\frac{\begin{array}{l}\mathtt{P},\Gamma \vdash_1 \mathtt{e_r} : \mathtt{t_r} \\ \mathtt{P},\Gamma \vdash_1 \mathtt{e_a} : \mathtt{t_a} \\ \mathcal{MS}\mathrm{ig}_1(\mathtt{P},\mathtt{t_r},\mathtt{m}) = \{\ \mathtt{t\ m(t}_a\ \mathtt{x)}\ \}\end{array}}{\mathtt{P},\Gamma \vdash_1 \mathtt{e_r.m(e_a)} : \mathtt{t}}$$

super-call

$$\frac{\begin{array}{l}\Gamma(\mathtt{this}) = \mathtt{t_r} \\ \mathtt{P},\Gamma \vdash_1 \mathtt{e_a} : \mathtt{t_a} \\ \mathcal{MS}\mathrm{ig}_1^{\mathtt{sup}}(\mathtt{P},\mathtt{t_r},\mathtt{m}) = \{\ \mathtt{t\ m(t}_a\ \mathtt{x)}\ \}\end{array}}{\mathtt{P},\Gamma \vdash_1 \mathtt{super.m(e_a)} : \mathtt{t}}$$

Fig. 8. *Chai*$_1$ Type Rules

For type checking we use a typing environment Γ which maps the receiver, `this`, and the method parameter, `x`, to a class name. The typing judgement $\mathtt{P},\Gamma \vdash_1 \mathtt{e} : \mathtt{t}$ means that in the context of program `P` and environment Γ, in the type system of *Chai*$_1$, the expression `e` has type `t`. Although in *Chai*$_1$ only classes can be types, the type rules in figure 8 mention types `t` rather than classes `cl`; this generality allows us to reuse these type rules for *Chai*$_2$.

The type rules, given in figure 8, *do not explicitly mention traits*, because traits have already been taken into account through $\mathcal{MS}\mathrm{ig}_1(\_,\_,\_)$ and $\mathcal{MS}\mathrm{ig}_1^{\mathtt{sup}}(\_,\_,\_)$. They are standard in all other respects: An expression of a certain type also has any of its supertypes (**subsumption**). The type of the formal parameter and receiver are looked up in the type environment (**var-this**). The creation of a new object has the type of that class, provided that the class is complete (**new**), while `null` has any type (**null**). The type of a method call is the return type of the function found by looking in the class of the first expression through $\mathcal{MS}\mathrm{ig}_1(\mathtt{P},\mathtt{cl},\mathtt{m})$, provided that the second expression has the type of the formal parameter type (**method-call**). Similarly for (**super-call**), where the method is looked-up in the superclass through $\mathcal{MS}\mathrm{ig}_1^{\mathtt{sup}}(\mathtt{P},\mathtt{cl},\mathtt{m})$.

$$
\frac{\begin{array}{l}
\mathtt{P}^{\mathtt{sup}}(\mathtt{cl}) = \mathtt{cl}' \\
\forall \mathtt{f}:\quad \mathtt{P}^{\mathtt{fld}}(\mathtt{cl},\mathtt{f}) = \mathtt{t} \quad \Longrightarrow \quad \mathcal{F}(\mathtt{P},\mathtt{cl}',\mathtt{f}) = \bot,\ \mathtt{P} \vdash_1 \mathtt{t} \diamond_{\text{type}} \\
\forall \mathtt{m}:\quad \mathtt{t}_0\ \mathtt{m}(\mathtt{t}_1\ \mathtt{x})\{\mathtt{e}\} \in \mathcal{M}(\mathtt{P},\mathtt{cl},\mathtt{m}) \quad \Longrightarrow \\
\qquad\qquad \mathtt{P} \vdash_1 \mathtt{t}_0 \diamond_{\text{type}} \\
\qquad\qquad \mathtt{P} \vdash_1 \mathtt{t}_1 \diamond_{\text{type}} \\
\qquad\qquad \mathtt{P},\ \mathtt{t}_1\ \mathtt{x},\ \mathtt{cl}\ \mathtt{this} \vdash_1 \mathtt{e} : \mathtt{t}_0 \\
\qquad\qquad \mathcal{M}(\mathtt{P},\mathtt{cl}',\mathtt{m}) = \emptyset\ \vee\ \mathcal{M}(\mathtt{P},\mathtt{cl}',\mathtt{m}) = \{\,\mathtt{t}_0\ \mathtt{m}(\mathtt{t}_1\ \mathtt{x})\ \{\ldots\}\,\}
\end{array}}{\mathtt{P} \vdash_1 \mathtt{cl}}
$$

$$
\frac{\text{for all classes } \mathtt{cl} \text{ defined in } \mathtt{P}:\quad \mathtt{P} \vdash_1 \mathtt{cl}}{\vdash_1 \mathtt{P}}
$$

Fig. 9. Well formed classes and programs in *Chai*$_1$

Note, that required methods do not play any rôle in *Chai*$_1$ type-checking (they do play a rôle in *Chai*$_2$ and *Chai*$_3$ type checking).

In figure 9 we define the notion of a well-formed *Chai*$_1$ class, *i.e.,* $\mathtt{P} \vdash_1 \mathtt{cl}$. A class `cl` is well-formed if:

1. Any field defined in that class has a valid type, and is not defined in its superclass `cl′`;
2. Any method defined in the class, or acquired though usage of a trait or inheritance from a superclass, has return and parameter type which are valid types, and, in a typing environment which maps `x` to the argument type $\mathtt{t}_1$ and `this` to `cl` (such an environment is written as $\mathtt{t}_1$ `x`, `cl` `this`), the method body has the declared return type $\mathtt{t}_0$. Additionally, if this method is present in the superclass, or any used traits, then it must be defined there with the same return type and parameter type.

The requirement 2. from above is very strong: It checks *all inherited and acquired* methods in a class - rather than just the methods defined in the class itself. Thus, a method defined in a trait will be type checked in all classes using that trait; this is unavoidable, because in *Chai*$_1$ method bodies cannot be checked in the traits.

A program is well-formed, $\vdash_1 \mathtt{P}$, if all its classes are well-formed. Note that the traits are not checked. Note also, that we do not require the inheritance hierarchy to be acyclic; although this is convenient, it is not necessary for soundness; in a program with cyclic inheritance, meaning can be given to lookup functions ($\mathcal{M}$, $\mathcal{M}^{\mathtt{orig}}$) through least fixed points.

3.6 Type Soundness

The judgement $\mathtt{P}, \sigma \vdash \mathtt{v} \lhd \mathtt{t}$ in figure 10 means that the value `v` agrees with the type `t`. In particular, if `v` is an address, it requires that the object at `v` belongs to a class `cl` which is a subtype of `t`, and for all fields defined in `cl`, the object contains values which agree with the types of the fields as declared in $\texttt{cl}$.[6] The judgement $\texttt{P},\Gamma \vdash_1 \sigma, \chi$ means that the objects in the heap χ agree with their classes, and belong to complete classes (*i.e.,* no conflicts), that the receiver object and argument value agree with their type as given in Γ, and that the class containing the method currently being executed ($\sigma(\texttt{this\_class})$) is the same as the type of the receiver in the type environment ($\Gamma(\texttt{this})$).

$$\frac{\texttt{P} \vdash_1 \texttt{t} \diamond_{\text{type}}}{\texttt{P},\chi \vdash_1 \texttt{null} \lhd \texttt{t}} \qquad \frac{\begin{array}{l}\chi(\iota) = [\![\, \texttt{cl} \,\|\, \dots \,]\!] \\ \texttt{P} \vdash_1 \texttt{cl} \leq \texttt{t} \\ \mathcal{F}(\texttt{P},\texttt{cl},\texttt{f}) = \texttt{t}' \Longrightarrow \texttt{P},\chi \vdash_1 \chi(\iota)(\texttt{f}) \lhd \texttt{t}'\end{array}}{\texttt{P},\chi \vdash_1 \iota \lhd \texttt{t}}$$

$$\frac{\begin{array}{l}\forall \iota: \quad \chi(\iota) = [\![\, \texttt{cl} \,\|\, \dots \,]\!] \quad \Longrightarrow \quad \texttt{P},\chi \vdash_1 \iota \lhd \texttt{cl}, \text{and } \texttt{P} \vdash_1 \texttt{cl} \diamond_{\text{cmpl}} \\ \texttt{P},\sigma \vdash_1 \sigma(\texttt{this}) \lhd \Gamma(\texttt{this}) \\ \texttt{P},\sigma \vdash_1 \sigma(\texttt{x}) \lhd \Gamma(\texttt{x}) \\ \sigma(\texttt{this\_class}) = \Gamma(\texttt{this})\end{array}}{\texttt{P},\Gamma \vdash_1 \sigma, \chi}$$

Fig. 10. Agreement in *Chai*$_1$

The following lemma is crucial in the proof of soundness, and guarantees that 1-2) the existence and types of fields and methods is preserved to subclasses, 3) that there are no more than one method signature per method in a superclass of a complete class (although there can be several method bodies, and 4) that if a method has a certain signature in a superclass $\texttt{cl}$', then method lookup in the subclass $\texttt{cl}$ will return a method body which type checks with this signature in the class $\texttt{cl}''$ which contains this method body (or inherits it from a trait).

Lemma 1. *If* $\vdash_1 \texttt{P}$ *and* $\texttt{P} \vdash_1 \texttt{cl} \leq \texttt{cl}'$ *then:*

1. $\mathcal{F}(\texttt{P},\texttt{cl}',\texttt{f}) = \texttt{t} \Longrightarrow \mathcal{F}(\texttt{P},\texttt{cl},\texttt{f}) = \texttt{t}$.
2. $\mathcal{MS}\text{ig}_1(\texttt{P},\texttt{cl}',\texttt{m}) \subseteq \mathcal{MS}\text{ig}_1(\texttt{P},\texttt{cl},\texttt{m})$.
3. $\texttt{P} \vdash_1 \texttt{cl} \diamond_{\text{cmpl}} \quad \Longrightarrow \quad |\, \mathcal{MS}\text{ig}_1(\texttt{P},\texttt{cl}',\texttt{m}) \,| \leq 1$.
4. $\texttt{t m(t}'\,\texttt{x)} \in \mathcal{MS}\text{ig}_1(\texttt{P},\texttt{cl}',\texttt{m}) \quad \Longrightarrow \quad \exists \texttt{cl}'', \texttt{e} :$
   - $\mathcal{M}^{\texttt{orig}}(\texttt{P},\texttt{cl},\texttt{m}) = \texttt{cl}'', \quad \texttt{t m(t}'\,\texttt{x)\{e\}} \in \mathcal{M}(\texttt{P},\texttt{cl},\texttt{m})$,
   - $\texttt{P} \vdash_1 \texttt{cl} \leq \texttt{cl}'' \quad \texttt{P},\texttt{t}'\,\texttt{x},\ \texttt{cl}''\,\texttt{this} \vdash_1 \texttt{e} : \texttt{t}$.

We can now prove soundness of the type system:

Theorem 2 (Type Soundness of *Chai*$_1$). *For program* $\texttt{P}$*, typing environment* Γ*, expression* $\texttt{e}$*, so that* $\texttt{super}$ *resolves without conflict in* $\texttt{e}$ *and* $\Gamma(\texttt{this\_class})$*, stack* σ*, heap* χ*, and type* $\texttt{t}$*:*

[6] Although the definition of $\texttt{P},\chi \vdash_1 \iota \lhd \texttt{t}$ is recursive, there exists an equivalent non-recursive definition for it.

If

$$\vdash_1 \texttt{P} \text{ and } \texttt{P}, \Gamma \vdash_1 \texttt{e : t} \text{ and } \texttt{P}, \Gamma \vdash_1 \sigma, \chi \text{ and } \texttt{e}, \sigma, \chi \leadsto_{\texttt{P}} \texttt{r}, \chi'$$

then:

$$\texttt{P}, \Gamma \vdash_1 \sigma, \chi' \quad \text{ and } \quad \texttt{P}, \chi' \vdash_1 \texttt{r} \lhd \texttt{t} \text{ or } \texttt{r}{=}\texttt{nllPntrExc}.$$

In other words, execution of well typed expressions preserves well formedness of the heap and stack, does not get stuck (since `r` is either a value or a null pointer exception), and if it returns a value, then this value is of the same type as the original expression.

4 The Language *Chai*$_2$

In *Chai*$_2$ we extended the remit of traits, so that they may be used as types. This has three important repercussions:

First, we can treat in a uniform way objects whose class uses a given trait, *e.g.,* we can write a stack for screenshapes, as in figure 2. Thus, traits support polymorphism, play the role of interfaces, and introduce multiple supertypes.

Second, we can typecheck traits in isolation, and therefore, we will be able to type check a method defined in a trait only once, rather than having to check it again in all the classes using that trait.

Third, we can take required methods into account, and can type check calls to required methods which do not have a method body in the receiver's class or trait. This is safe, because we allow object creation only for *complete* classes, and *Chai*$_2$ complete classes are those that provide method bodies for all required methods.

4.1 *Chai*$_2$ Syntax and Operational Semantics

The only difference between the syntax of *Chai*$_2$ and that of *Chai*$_1$ is that *Chai*$_2$ allows traits to be types, i.e:

$$\textit{type} ::= \texttt{cl} \mid \texttt{tr}$$

The operational semantics of *Chai*$_2$ is identical to that of *Chai*$_1$.

4.2 Required Methods

We fist define *indirect use* of traits, where $\mathcal{U}se^*(\texttt{P},\texttt{tr})$ collects the transitive closure of the traits used in `tr`, and $\mathcal{U}se^*(\texttt{P},\texttt{cl})$ collects all traits indirectly used by traits used in `cl`, or in `cl`'s superclasses.

$$\begin{aligned}\mathcal{U}se^*(\texttt{P},\texttt{tr}) &= \textstyle\bigcup_{\texttt{tr}' \in \texttt{P}^{\texttt{use}}(\texttt{tr})} \mathcal{U}se^*(\texttt{P},\texttt{tr}') \ \cup \ \{\ \texttt{tr}\ \}\\ \mathcal{U}se^*(\texttt{P},\texttt{cl}) &= \textstyle\bigcup_{\texttt{P} \vdash_1 \texttt{cl} \leq \texttt{cl}',\ \texttt{tr} \in \texttt{P}^{\texttt{use}}(\texttt{cl}')} \mathcal{U}se^*(\texttt{P},\texttt{tr})\end{aligned}$$

A trait `tr` may define a list of *required* methods. A second trait `tr′` which uses `tr` inherits `tr`'s required methods and may add new requirements of its own, through explicit requirements or through exclusion. A class using `tr` inherits the requirements of `tr`.

$$\begin{aligned}
\mathcal{MR}\mathrm{eq}(\mathtt{P},\mathtt{tr},\mathtt{m}) &= \textstyle\bigcup_{\mathtt{tr}'\in\mathcal{U}se^*(\mathtt{P},\mathtt{tr})} \mathtt{P}^{\mathtt{req}}(\mathtt{tr}') \cup \\
&\quad \textstyle\bigcup_{\mathtt{tr}'\in\mathcal{U}se^*(\mathtt{P},\mathtt{tr})} \{\ \mathtt{t\ m(t'\,x)} \mid \exists \mathtt{tr}'' :\ (\mathtt{tr}'',\mathtt{m}) \in \mathtt{P}^{\mathtt{excl}}(\mathtt{tr}'), \\
&\qquad \mathtt{t\ m(t'\,x)} \in \mathcal{MS}\mathrm{ig}_1(\mathtt{P},\mathtt{tr}'',\mathtt{m}) \cup \mathcal{MR}\mathrm{eq}(\mathtt{P},\mathtt{tr}'',\mathtt{m})\ )\} \\
\mathcal{MR}\mathrm{eq}(\mathtt{P},\mathtt{cl},\mathtt{m}) &= \textstyle\bigcup_{\mathtt{tr}\in\mathcal{U}se^*(\mathtt{P},\mathtt{cl})} \mathcal{MR}\mathrm{eq}(\mathtt{P},\mathtt{tr},\mathtt{m}) \\
\mathcal{MR}\mathrm{eq}^{\mathrm{sup}}(\mathtt{P},\mathtt{tr},\mathtt{m}) &= \textstyle\bigcup_{\mathtt{tr}'\in\mathcal{U}se^*(\mathtt{P},\mathtt{tr})} \mathtt{P}^{\mathtt{req\_sup}}(\mathtt{tr}') \\
\mathcal{MR}\mathrm{eq}^{\mathrm{sup}}(\mathtt{P},\mathtt{cl},\mathtt{m}) &= \textstyle\bigcup_{\mathtt{tr}\in\mathcal{U}se^*(\mathtt{P},\mathtt{cl})} \mathcal{MR}\mathrm{eq}^{\mathrm{sup}}(\mathtt{P},\mathtt{tr},\mathtt{m})
\end{aligned}$$

Note, that it is possible for a signature to be required in a trait `tr`, and for the trait to have a method body for this signature. Similarly for classes.

A class which is complete in the sense of *Chai*$_1$, and where all required methods have a body is complete for *Chai*$_2$:

$$\frac{\begin{array}{l}
\forall \mathtt{m}:\ \mathtt{t\ m(t'\,x)} \in \mathcal{MR}\mathrm{eq}(\mathtt{P},\mathtt{cl},\mathtt{m}) \implies \exists \mathtt{e}:\ \mathtt{t\ m(t'\,x)\{e\}} \in \mathcal{M}(\mathtt{P},\mathtt{cl},\mathtt{m}) \\
\forall \mathtt{m}:\ |\mathcal{M}(\mathtt{P},\mathtt{cl},\mathtt{m})| \leq 1 \\
\forall \mathtt{m},\mathtt{cl}'\quad \mathtt{P} \vdash_1 \mathtt{cl} \leq \mathtt{cl}',\ ...\{\mathtt{e}\} \in \mathcal{M}(\mathtt{P},\mathtt{cl}',\mathtt{m}) \implies \\
\qquad \mathtt{super}\text{ resolves without conflict in } \mathtt{e}\text{ and}\mathtt{cl}'
\end{array}}{\mathtt{P} \vdash_2 \mathtt{cl} \diamond_{\mathrm{cmpl}}}$$

Thus, a complete subclass of `t` will provide a method body for any method required by `t`. The function $\mathcal{MS}\mathrm{ig}_2(\mathtt{P},\mathtt{t},\mathtt{m})$ returns the signatures of the method that will be provided for `m` by a complete subclass of `t`, while the function $\mathcal{MS}\mathrm{ig}_2^{\mathtt{sup}}(\mathtt{P},\mathtt{t},\mathtt{m})$ returns the signatures of all methods that will be provided in the superclass of a complete subclass of `t`:

$$\begin{aligned}
\mathcal{MS}\mathrm{ig}_2(\mathtt{P},\mathtt{tr},\mathtt{m}) &= \mathcal{MS}\mathrm{ig}_1(\mathtt{P},\mathtt{tr},\mathtt{m}) \cup \mathcal{MR}\mathrm{eq}(\mathtt{P},\mathtt{tr},\mathtt{m}) \\
\mathcal{MS}\mathrm{ig}_2(\mathtt{P},\mathtt{cl},\mathtt{m}) &= \mathcal{MS}\mathrm{ig}_1(\mathtt{P},\mathtt{cl},\mathtt{m}) \cup \mathcal{MR}\mathrm{eq}(\mathtt{P},\mathtt{cl},\mathtt{m}) \\
\mathcal{MS}\mathrm{ig}_2^{\mathtt{sup}}(\mathtt{P},\mathtt{tr},\mathtt{m}) &= \mathcal{MS}\mathrm{ig}_1^{\mathtt{sup}}(\mathtt{P},\mathtt{cl},\mathtt{m}) \cup \mathcal{MR}\mathrm{eq}^{\mathrm{sup}}(\mathtt{P},\mathtt{tr},\mathtt{m}) \\
\mathcal{MS}\mathrm{ig}_2^{\mathtt{sup}}(\mathtt{P},\mathtt{cl},\mathtt{m}) &= \mathcal{MS}\mathrm{ig}_1^{\mathtt{sup}}(\mathtt{P},\mathtt{cl},\mathtt{m}) \cup \mathcal{MR}\mathrm{eq}^{\mathrm{sup}}(\mathtt{P},\mathtt{cl},\mathtt{m})
\end{aligned}$$

Notice, that $\mathtt{t} \in \mathtt{P}^{\mathtt{use}}(\mathtt{t}')$ implies that $\mathcal{MS}\mathrm{ig}_2(\mathtt{P},\mathtt{t},\mathtt{m}) \subseteq \mathcal{MS}\mathrm{ig}_2(\mathtt{P},\mathtt{t}',\mathtt{m})$ for all `m`, and class or trait `t`, and `t'`.

4.3 Type System

As we define in figure 11, a class or trait is a subtype of any trait that it uses - possibly indirectly. Thus, a class `cl` or trait `tr` that uses a trait `tr'` is a subtype of `tr'`, even if `tr` *requires more* methods than `tr'`. This may seem surprising, but it is safe for the following reason: even though traits are types, the runtime entities (i.e. the objects) will belong to *complete* classes, which, by definition, provide a method body for any required method. The ensuing subtype relationship is transitive.

Note that $\mathtt{P} \vdash_2 \mathtt{t}' \leq \mathtt{t}$ implies that $\mathcal{MS}\mathrm{ig}_2(\mathtt{P},\mathtt{t},\mathtt{m}) \subseteq \mathcal{MS}\mathrm{ig}_2(\mathtt{P},\mathtt{t}',\mathtt{m})$ - we could have defined subtypes in a structural, rather than a nominal way using the above property.

In *Chai*$_2$ traits can be types, therefore in *Chai*$_2$ typing environments may map `this`, and `x` to a trait or a class. The typing rules are the same as those for *Chai*$_1$, with three exceptions. First, the **subsumption** rule uses the new

$$\frac{\mathtt{P} \vdash \mathtt{tr} \diamond_{\text{class}}}{\mathtt{P} \vdash_2 \mathtt{cl} \diamond_{\text{type}}} \qquad \frac{\mathtt{P} \vdash \mathtt{tr} \diamond_{\text{trait}}}{\mathtt{P} \vdash_2 \mathtt{tr} \diamond_{\text{type}}}$$

$$\frac{\mathtt{P} \vdash_1 \mathtt{cl} \leq \mathtt{cl}'}{\mathtt{P} \vdash_2 \mathtt{cl} \leq \mathtt{cl}'} \qquad \frac{\mathtt{tr} \in \mathcal{U}se^*(\mathtt{P},\mathtt{cl})}{\mathtt{P} \vdash_2 \mathtt{cl} \leq \mathtt{tr}} \qquad \frac{\mathtt{tr} \in \mathcal{U}se^*(\mathtt{P},\mathtt{tr}')}{\mathtt{P} \vdash_2 \mathtt{tr}' \leq \mathtt{tr}}$$

Fig. 11. Types and Subtypes in *Chai*$_2$

subtype relation $\mathtt{P} \vdash_2 \mathtt{t}' \leq \mathtt{t}$. Second, the rules **method-call** and **super-call** take the required method into account, *i.e.,* use $\mathcal{MS}\text{ig}_2(\mathtt{P},\mathtt{t},\mathtt{m})$ and $\mathcal{MS}\text{ig}_2^{\mathtt{sup}}(\mathtt{P},\mathtt{t},\mathtt{m})$. Third, the rule **new** requires the class to be complete according to $\mathtt{P} \vdash_2 \mathtt{cl} \diamond_{\text{cmpl}}$.

A trait $\mathtt{tr}$ is well formed, *i.e.,* $\mathtt{P} \vdash_2 \mathtt{tr}$ in figure 12, if the methods *directly defined* in that trait are well-typed, and have the same signature as any method with the same identifier acquired from a used trait.

A class $\mathtt{cl}$ is well formed, *i.e.,* $\mathtt{P} \vdash_2 \mathtt{cl}$, if the fields in that class have well-formed types; and if the methods *directly defined* in that class are well-typed, and have the same signature as any method acquired from a used trait, or inherited from a superclass. A program is well formed, if all its classes and traits are well formed.

Notice that, to establish $\mathtt{P} \vdash_2 \mathtt{t}$ we only check the methods *directly defined* in class or trait $\mathtt{t}$; (we use $\mathtt{P}^{\mathtt{mth}}(\mathtt{cl},\mathtt{m})$ - as opposed to $\mathcal{M}(\mathtt{P},\mathtt{cl},\mathtt{m})$ in *Chai*$_1$). Also, $\mathtt{P} \vdash_1 \mathtt{t}$ does not *imply* $\mathtt{P} \vdash_2 \mathtt{t}$, and nor does $\mathtt{P} \vdash_2 \mathtt{t}$ imply $\mathtt{P} \vdash_1 \mathtt{t}$.

4.4 Type Soundness

In *Chai*$_2$ we retain the definition of agreement between objects and classes from figure 10, but use the subtype relation $\mathtt{P} \vdash_2 \mathtt{t} \leq \mathtt{t}'$, and the definition of complete classes $\mathtt{P} \vdash_2 \mathtt{cl} \diamond_{\text{cmpl}}$ from this section.

Thus, we were able to give "uniform" definitions of *Chai*$_1$ and *Chai*$_2$, and distill their similarities and differences.

The following lemma is the counterpart to lemma 1; the difference is that here we talk of types (and thus also of traits) rather than just of classes, we use the *Chai*$_2$ subtype relationship with also incorporates traits usage, and in the *Chai*$_2$ signature lookup function we also take the requirements into account.

Lemma 3. *If* $\vdash_1 \mathtt{P}$ *and classes* $\mathtt{cl}$, $\mathtt{cl}'$ *and types* $\mathtt{t}$ *and* $\mathtt{t}'$, *with* $\mathtt{P} \vdash_2 \mathtt{cl} \leq \mathtt{cl}'$, $\mathtt{P} \vdash_2 \mathtt{t} \leq \mathtt{t}'$, *and* $\mathtt{P} \vdash_2 \mathtt{cl} \leq \mathtt{t}'$, *then*:

1. $\mathcal{MS}\text{ig}_2(\mathtt{P},\mathtt{t}',\mathtt{m}) \subseteq \mathcal{MS}\text{ig}_2(\mathtt{P},\mathtt{t},\mathtt{m})$.
2. $\mathtt{P} \vdash_2 \mathtt{cl} \diamond_{\text{cmpl}}, \implies |\, \mathcal{MS}\text{ig}_2(\mathtt{P},\mathtt{t}',\mathtt{m}) \,| \leq 1$.
3. $\mathtt{P},\mathtt{t_a}\ \mathtt{x},\ \mathtt{t}'\ \mathtt{this} \vdash_2 \mathtt{e} : \mathtt{t}'' \quad \Longrightarrow \quad \mathtt{P},\mathtt{t_a}\ \mathtt{x},\ \mathtt{t}\ \mathtt{this} \vdash_2 \mathtt{e} : \mathtt{t}''$.
4. $\mathtt{P} \vdash_2 \mathtt{cl} \diamond_{\text{cmpl}},\ \mathtt{t}''\ \mathtt{m}(\mathtt{t}'''\ \mathtt{x}) \in \mathcal{MS}\text{ig}_2(\mathtt{P},\mathtt{t}',\mathtt{m}) \quad \Longrightarrow \quad \exists \mathtt{cl}'', \mathtt{e} :$
   - $\mathcal{M}^{\mathtt{orig}}(\mathtt{P},\mathtt{cl},\mathtt{m}) = \mathtt{cl}'', \quad \mathtt{t}''\ \mathtt{m}(\mathtt{t}'''\ \mathtt{x})\{\mathtt{e}\} \in \mathcal{M}(\mathtt{P},\mathtt{cl}'',\mathtt{m})$,
   - $\mathtt{P} \vdash_1 \mathtt{cl} \leq \mathtt{cl}'' \quad \mathtt{P},\mathtt{t}'''\ \mathtt{x},\ \mathtt{cl}''\ \mathtt{this} \vdash_2 \mathtt{e}'' : \mathtt{t}''$.

$$\frac{\begin{array}{l}\forall \mathtt{m} : \mathtt{t}_0\ \mathtt{m}(\mathtt{t}_1\ \mathtt{x})\{\mathtt{e}\} \in \mathtt{P}^{\mathtt{mth}}(\mathtt{P}) \Longrightarrow \\ \qquad \mathtt{P} \vdash_2 \mathtt{t}_0 \diamond_{\text{type}} \\ \qquad \mathtt{P} \vdash_2 \mathtt{t}_1 \diamond_{\text{type}} \\ \qquad \mathtt{P},\ \mathtt{t}_1\ \mathtt{x},\ \mathtt{tr}\ \mathtt{this} \vdash_2 \mathtt{e}\ :\ \mathtt{t}_0 \\ \qquad \forall \mathtt{tr}' : \mathtt{tr}' \in \mathtt{P}^{\mathtt{use}}(\mathtt{tr}) \Longrightarrow \\ \qquad\qquad \mathcal{MS}\mathrm{ig}_2(\mathtt{P},\mathtt{tr}',\mathtt{m}) = \emptyset\ \vee\ \mathcal{MS}\mathrm{ig}_2(\mathtt{P},\mathtt{tr}',\mathtt{m})\{\mathtt{t}_0\ \mathtt{m}(\mathtt{t}_1\ \mathtt{x})\}\end{array}}{\mathtt{P} \vdash_2 \mathtt{tr}}$$

$$\frac{\begin{array}{l}\mathtt{cl}' = \mathtt{P}^{\mathtt{sup}}(\mathtt{cl}) \\ \forall \mathtt{f}:\ \ \mathtt{P}^{\mathtt{fld}}(\mathtt{cl},\mathtt{f}) = \mathtt{t} \Longrightarrow \mathtt{P} \vdash_2 \mathtt{t} \diamond_{\text{type}}\ ,\ \mathcal{F}(\mathtt{P},\mathtt{cl}',\mathtt{f}) = \bot \\ \forall \mathtt{m}:\ \ \mathtt{t}_0\ \mathtt{m}(\mathtt{t}_1\ \mathtt{x})\{\mathtt{e}\} \in \mathtt{P}^{\mathtt{mth}}(\mathtt{cl},\mathtt{m}) \Longrightarrow \\ \qquad \mathtt{P} \vdash_2 \mathtt{t}_0 \diamond_{\text{type}} \\ \qquad \mathtt{P} \vdash_2 \mathtt{t}_1 \diamond_{\text{type}} \\ \qquad \mathtt{P},\ \mathtt{t}_1\ \mathtt{x},\ \mathtt{cl}\ \mathtt{this} \vdash_2 \mathtt{e}\ :\ \mathtt{t}_0 \\ \qquad \mathcal{MS}\mathrm{ig}_2(\mathtt{P},\mathtt{cl}',\mathtt{m}) = \emptyset\ \vee\ \mathcal{MS}\mathrm{ig}_2(\mathtt{P},\mathtt{cl}',\mathtt{m}) = \{\mathtt{t}_0\ \mathtt{m}(\mathtt{t}_1\ \mathtt{x})\} \\ \qquad \forall \mathtt{tr} \in \mathtt{P}^{\mathtt{use}}(\mathtt{cl}) : \mathcal{MS}\mathrm{ig}_2(\mathtt{P},\mathtt{tr},\mathtt{m}) = \emptyset\ \vee\ \mathcal{MS}\mathrm{ig}_2(\mathtt{P},\mathtt{tr},\mathtt{m}) = \{\mathtt{t}_0\ \mathtt{m}(\mathtt{t}_1\ \mathtt{x})\}\end{array}}{\mathtt{P} \vdash_2 \mathtt{cl}}$$

$$\frac{\begin{array}{l}\text{for all classes } \mathtt{cl} \text{ defined in } \mathtt{P}:\ \ \mathtt{P} \vdash_2 \mathtt{cl} \\ \text{for all traits } \mathtt{tr} \text{ defined in } \mathtt{P}:\ \ \mathtt{P} \vdash_2 \mathtt{tr}\end{array}}{\vdash_2 \mathtt{P}}$$

Fig. 12. Well-formed traits, classes and programs in *Chai*$_2$

With the above lemma we can prove soundness for the type system of *Chai*$_2$:

Theorem 4 (Type Soundness of *Chai*$_2$**).** *For any program* $\mathtt{P}$*, environment* Γ*, expression* $\mathtt{e}$ *with* `super` *resolves without conflict in* $\mathtt{e}$ *and* $\Gamma(\mathtt{this\_class})$*, stack* σ*, type* $\mathtt{t}$*, where* $\vdash_2 \mathtt{P}$*, and* $\mathtt{P},\Gamma \vdash_2 \mathtt{e}\ :\ \mathtt{t}$ *and* $\mathtt{P},\Gamma \vdash_2 \sigma,\ \chi$ *and* $\mathtt{e},\ \sigma,\ \chi \leadsto_{\mathtt{P}} \mathtt{r}, \chi'$*:*

$$\mathtt{P},\Gamma \vdash_2 \sigma,\ \chi' \qquad \textit{and} \qquad \mathtt{P},\chi \vdash_2 \mathtt{r} \lhd \mathtt{t}\ \textit{or}\ \mathtt{r}{=}\mathtt{nllPntrExc}.$$

5 The Language *Chai*$_3$

Chai$_3$ introduces *dynamic trait substitution.* Since traits specify pure behaviour, it should be possible to substitute one trait for another at runtime in order to change the behaviour of an object. Outwardly, the interface of the object would remain the same, providing the same fields and methods, but internally the implementation of various methods could be altered.

Although the idea of objects changing behaviour at runtime (dynamic object re-classification) has been presented in several different forms[7, 22], the only time this concept has been explored in the existing literature on traits[7] is relation

[7] The authors of [18] mention using traits to dynamically change object behaviour as an element of future work.

to the object-based language SELF[1, 22], where dynamic changes in behaviour can be obtained by changing which object acts as the parent of the current object. We present a mechanism supporting dynamic traits inspired by the ideas from SELF, but in a class-based language.

5.1 Example

Consider a graphical windowing system: A window in this system may be an `OpenedWindow` or an `IconifiedWindow`. In each state the window will behave differently, and a window may change between these two states at any time.

To implement this in traditional Object Oriented programming, we would need to use wrappers, or some form of the state pattern.

Using dynamic substitution of traits, we can offer a more elegant, and direct solution: we define a class `Window`, and two traits `TOpened` and `TIconified`, where `TOpened` and `TIconified` provide and require the same sets of method signatures, but provide different implementations of the methods and so different behaviour. We define the class `Window` as `class Window uses TOpened ...` (the window begins in the opened state). Then, for a Window object `w` (`Window w = new Window();`) we can change to the iconified state using the statement `w<TOpened ↦TIconified>`. This will result in the substitution of the trait `TIconified` for the trait `TOpened` inside the object `w`.

Since the class `Window` was declared as using the trait `TOpened`, the label `TOpened` becomes a "placeholder" for that trait used by `Window`, and a trait "compatible" with `TOpened` can be substituted for `TOpened` at any time. We use the label `TOpened` in all further substitutions for that trait "placeholder" of `w`. For example, to switch back to the original behaviour of `w`, we write `w<TOpened ↦TOpened>` (and *not*, as might be imagined, `w<TIconified ↦TOpened>`).

5.2 *Chai*$_3$ Syntax and Operational Semantics

We extended the syntax of expressions to allow trait substitution.

$$\textit{exp} ::= \ \textit{exp}{<}\ \texttt{tr} \mapsto \texttt{tr} > \ | \ \ ...$$

Resolving Method Calls. Consider the program given in figure 13. If we create an object of class `C`, e.g `C x = new C`, then obviously executing `x.m1()` will return the value 3, and executing `x.m2()` will also return the value 3.

If we execute `x` < `TrtB` ↦ `TrtB2` > followed by `x.m1()`, then the version of `m1` provided by `TrtA` will be used, since the method `m1` was originally provided to class `C` by trait `TrtA`, and no trait has replaced `TrtA` in `c`.

If we execute `x` < `TrtB` ↦ `TrtB2` > followed by `x.m2()`, then the situation is more complex. Obviously, the method `m2` defined in `TrtB2` will be executed (since `TrtB` originally provided `m2`, and `TrtB` has been replaced by `TrtB2`). However, there are three possibilities for the binding of `m1` from within the body of `m2`:

1. The version of `m1` from `TrtA` will be used; because invoking a method from within a trait should have the same semantics as invoking it from within the class using the trait. Thus, we resolve methods based on the flattened version of the class using the traits.

```
trait TrtA { int m1() { 3 } }          trait TrtB2 {
                                          int m2() { this.m1() }
trait TrtB {                              int m1() { 5 }
   requires { int m1(); }              }
   int m2() { this.m1() }
}                                      class C uses TrtA,TrtB { }
```

Fig. 13. Resolving Method Calls in *Chai*$_3$

2. The version of `m1` from `TrtB2` will be used; because the methods in `TrtB2` are interrelated, it is likely that the implementor of `TrtB2` intended the call to `m1` to resolve to the method in `TrtB2`. Thus, we resolve methods based on the trait in which the call was found.
3. The situation is illegal; *i.e.,* trait `TrtB2` cannot be substituted for trait `TrtB` because it creates this "ambiguity" regarding the definition of method `m1`.

In this paper, we chose option 1 from above, because of its close relationship to the flattening property which is a crucial element of Traits philosophy.

Object Representation. Substitution of traits at runtime is on a per-object basis (rather than a per-class basis). This means that while the list of traits used by any class remains constant, for every object of that class, each used trait may be associated with some (possibly different) trait. Therefore, we extend the representation objects from figure 5 with a list of trait substitutions that have been made to the object.

$$\textbf{object} = \{\ [\![\ \mathtt{cl} \parallel \mathtt{f}_1 : \mathtt{v}_1, \ldots \mathtt{f}_\mathtt{r} : \mathtt{v}_\mathtt{r} \parallel \mathtt{tr}_1 : \mathtt{tr}'_1, \ldots \mathtt{tr}_\mathtt{n} : \mathtt{tr}'_\mathtt{n}\ ]\!]\quad | \quad \mathtt{cl}, \mathtt{f}_1, \ldots \mathtt{f}_\mathtt{r}, \mathtt{tr}_1, \ldots \mathtt{tr}_\mathtt{n}, \mathtt{tr}'_1, \ldots \mathtt{tr}'_\mathtt{n} \text{ identifiers; } \mathtt{v}_1, \ldots \mathtt{v}_\mathtt{r} \in \textbf{val}\ \}$$

To access and update these trait substitutions for an object $o = [\![\ \mathtt{cl} \parallel \ldots \parallel \mathtt{tr}_1 : \mathtt{tr}'_1, \ldots \mathtt{tr}_\mathtt{n} : \mathtt{tr}'_\mathtt{n}\ ]\!]$, we define *trait lookup* $o(\mathtt{tr})$ which finds the current substitution for a given trait name, and *object mutation* $o[\mathtt{tr} \mapsto \mathtt{tr}']$ which replaces the trait named $\mathtt{tr}$ by $\mathtt{tr}'$.

$$o(\mathtt{tr}) \quad = \begin{cases} \mathtt{tr}'_\mathtt{k} & \text{if } \mathtt{tr} = \mathtt{tr}_\mathtt{k} \text{ for some } \mathtt{k} \in \mathtt{1}, \ldots \mathtt{n} \\ \bot & \text{otherwise.} \end{cases}$$

$$o[\mathtt{tr} \mapsto \mathtt{tr}'] = \begin{cases} [\![\ \mathtt{cl} \parallel \ldots \parallel \mathtt{tr}_1 : \mathtt{tr}'_1 \ldots \mathtt{tr}_\mathtt{k} : \mathtt{tr}' \ldots \mathtt{tr}_\mathtt{n} : \mathtt{tr}'_\mathtt{n}\ ]\!] & \text{for } \mathtt{tr} = \mathtt{tr}_k, \mathtt{k} \in \mathtt{1}, \ldots, \mathtt{n} \\ \bot \quad \text{otherwise.} \end{cases}$$

Runtime Method Lookup and Operational Semantics. Trait substitutions must be taken into account for method call. The function $\mathcal{M}_3$ finds the appropriate method body, taking both the class of the object, and the object itself into account – the latter is needed, in order to find the traits that have replaced the original ones. $\mathcal{M}_3$ first determines which class or trait name is "responsible" for the corresponding method through $\mathcal{M}_3^{\text{resp}}(\mathtt{P},\mathtt{cl},\mathtt{m})$, which first searches the current class, then the used traits, and then continues with the superclass. If $\mathcal{M}_3^{\text{resp}}(\mathtt{P},\mathtt{cl},\mathtt{m})$ is a class $\mathtt{cl}'$ then the method body is found directly in $\mathtt{cl}'$. If $\mathcal{M}_3^{\text{resp}}(\mathtt{P},\mathtt{cl},\mathtt{m})$ is a trait $\mathtt{tr}$ then the method body is found in trait $\mathtt{tr}'$, which replaces $\mathtt{tr}$ in the current object (*i.e.,* $o(\mathtt{tr}) = \mathtt{tr}'$).

$$\mathcal{M}_3^{\text{resp}}(\mathtt{P},\mathtt{tr},\mathtt{m}) = \begin{cases} \{\ \mathtt{tr}\ \} & \text{if } \mathtt{P}^{\mathtt{mth}}(\mathtt{tr},\mathtt{m}) \neq \emptyset \\ \bigcup_{\mathtt{tr}' \in \mathtt{P}^{\mathtt{use}}(\mathtt{tr})} \mathcal{M}_3^{\text{resp}}(\mathtt{P},\mathtt{tr}',\mathtt{m}) & \text{otherwise.} \end{cases}$$

$$\mathcal{M}_3^{\text{resp}}(\mathtt{P},\mathtt{cl},\mathtt{m}) = \begin{cases} \{\ \mathtt{cl}\ \} & \text{if } \mathtt{P}^{\mathtt{mth}}(\mathtt{cl},\mathtt{m}) \neq \emptyset \\ \mathit{Trts} & \text{where } \mathit{Trts} = \bigcup_{\mathtt{tr} \in \mathtt{P}^{\mathtt{use}}(\mathtt{cl})} \mathcal{M}_3^{\text{resp}}(\mathtt{P},\mathtt{tr},\mathtt{m}) \\ & \text{if } \mathit{Trts} \neq \emptyset = \mathtt{P}^{\mathtt{mth}}(\mathtt{cl},\mathtt{m}) \\ \mathcal{M}_3^{\text{resp}}(\mathtt{P},\mathtt{P}^{\mathtt{sup}}(\mathtt{cl}),\mathtt{m}) & \text{otherwise.} \end{cases}$$

$$\mathcal{M}_3(\mathtt{P},\mathtt{cl},\mathtt{o},\mathtt{m}) = \begin{cases} \mathtt{P}^{\mathtt{mth}}(\mathtt{cl}',\mathtt{m}) & \text{if } \mathcal{M}_3^{\text{resp}}(\mathtt{P},\mathtt{cl},\mathtt{m}) = \{\mathtt{cl}'\} \\ \mathtt{P}^{\mathtt{mth}}(\mathtt{tr}',\mathtt{m}) & \text{if } \mathcal{M}_3^{\text{resp}}(\mathtt{P},\mathtt{cl},\mathtt{m}) = \{\mathtt{tr}\},\ \text{and } o(\mathtt{tr}) = \mathtt{tr}' \\ \bot & \text{otherwise.} \end{cases}$$

A class $\mathtt{cl}$ is complete in *Chai*$_3$ if it provides a method body for any required method, if there are no conflicts for any superclass (this simplifies the treatment of `super`), and if $\mathcal{M}_3^{\text{resp}}(\mathtt{P},\mathtt{cl},\mathtt{m})$ is empty or a singleton.

$$\frac{\begin{array}{l} \forall \mathtt{m}:\ \mathtt{t}\ \mathtt{m}(\mathtt{t}'\,\mathtt{x}) \in \mathcal{MR}\text{eq}(\mathtt{P},\mathtt{cl},\mathtt{m}) \implies \exists \mathtt{e}:\ \mathtt{t}\ \mathtt{m}(\mathtt{t}'\,\mathtt{x})\{\mathtt{e}\} \in \mathcal{M}(\mathtt{P},\mathtt{cl},\mathtt{m}) \\ \forall \mathtt{m},\mathtt{cl}' \quad \mathtt{P} \vdash_1 \mathtt{cl} \leq \mathtt{cl}',\ |\mathcal{M}(\mathtt{P},\mathtt{cl}',\mathtt{m})| \leq 1 \\ \forall \mathtt{m}:\ \ |\,\mathcal{M}_3^{\text{resp}}(\mathtt{P},\mathtt{cl},\mathtt{m})\,| \leq 1 \end{array}}{\mathtt{P} \vdash_3 \mathtt{cl}\ \diamond_{\text{cmpl}}}$$

The operational semantics of *Chai*$_3$ differs from that of *Chai*$_1$ and *Chai*$_2$ in the handling of mutation, object creation, and method call, therefore, we extend the semantics from figure 6. A mutate expression substitutes one trait by another (**mutate**). Object creation initializes the fields *and* the list of trait substitutions for new objects through the identity substitution, i.e. associates all traits with themselves (**new**).

mutate

$$\frac{\begin{array}{l} \mathtt{e},\sigma,\chi \rightsquigarrow_{\mathtt{P}} \iota,\chi'' \\ \chi' = \chi''[\iota \mapsto \chi''(\iota)[\mathtt{tr} \mapsto \mathtt{tr}']] \end{array}}{\mathtt{e} < \mathtt{tr} \mapsto \mathtt{tr}' >,\sigma,\chi \rightsquigarrow_{\mathtt{P}} \iota,\chi'}$$

new

$$\frac{\begin{array}{l} \mathcal{F}_{\text{s}}(\mathtt{P},\mathtt{cl}) = \{\ \mathtt{f}_1,\ldots,\mathtt{f}_{\mathtt{r}}\ \} \\ \{\ \mathtt{tr}_1,\ldots\mathtt{tr}_{\mathtt{n}}\ \} = \mathcal{U}se^*(\mathtt{P},\mathtt{cl}) \\ \iota \text{ is new in } \chi \\ o = [\![\ \mathtt{cl}\ \|\ \mathtt{f}_1\!:\!\mathtt{null}\ldots\mathtt{f}_{\mathtt{n}}\!:\!\mathtt{null}\ \| \\ \qquad\qquad \mathtt{tr}_1\!:\!\mathtt{tr}_1\ldots\mathtt{tr}_{\mathtt{n}}\!:\!\mathtt{tr}_{\mathtt{n}}\ ]\!] \end{array}}{\mathtt{new}\ \mathtt{cl},\sigma,\chi \rightsquigarrow_{\mathtt{P}} \iota,\chi[\iota \mapsto \mathtt{o}]}$$

In method call we use the new method lookup function $\mathcal{M}_3(\mathtt{P},\mathtt{c},\mathtt{o},\mathtt{m})$ (**method-call**). Thus, if a trait is used in class `cl` through two different paths (*e.g.,* used by `cl`, and also by $\mathtt{cl}'$, where $\mathtt{cl}'$ is `cl`'s superclass), then mutation of the trait will affect the behaviour of its methods regardless of the path used to access the object (*e.g.,* as a value of type `cl`, or $\mathtt{cl}'$) - this is consistent with the flattening property. On the other hand, if a trait $\mathtt{tr}$ which uses trait $\mathtt{tr}'$ is replaced by $\mathtt{tr}''$, then only the methods directly provided by $\mathtt{tr}$ will be looked up in trait $\mathtt{tr}''$; the ones that were inherited by $\mathtt{tr}'$ will remain unaffected. This is, in some sense, inconsistent with the flattening property, and in further work we would like to investigate alternatives.

method-call

$$\frac{\begin{array}{l}\mathtt{e_r},\ \sigma,\ \chi \rightsquigarrow_{\mathtt{P}} \iota, \chi_0 \\ \mathtt{e_a},\ \sigma,\ \chi_0 \rightsquigarrow_{\mathtt{P}} \mathtt{v_1}, \chi_1 \\ \chi_1(\iota) = [\![\, \mathtt{cl} \,\|\, \dots \,]\!] \\ \mathcal{M}_3(\mathtt{P}, \mathtt{cl}, \chi_1(\iota), \mathtt{m}) = \\ \qquad\qquad \{\ \mathtt{t\ m(t'\ x)}\ \{\ \mathtt{e}\ \}\ \} \\ \mathcal{M}^{\mathtt{orig}}(\mathtt{P}, \mathtt{cl}, \mathtt{m}) = \mathtt{cl}' \\ \sigma' = (\iota, \mathtt{v_1}, \mathtt{cl}') \\ \mathtt{e},\ \sigma',\ \chi_1 \rightsquigarrow_{\mathtt{P}} \mathtt{v}, \chi'\end{array}}{\mathtt{e_r.m(e_a)},\ \sigma,\ \chi \rightsquigarrow_{\mathtt{P}} \mathtt{v}, \chi'}$$

super-call

$$\frac{\begin{array}{l}\mathtt{e_a},\ \sigma,\ \chi \rightsquigarrow_{\mathtt{P}} \mathtt{v_1}, \chi_1 \\ \sigma(\mathtt{this\_class}) = \mathtt{cl} \\ \mathtt{P}^{\mathtt{sup}}(\mathtt{cl}) = \mathtt{cl}'' \\ \mathcal{M}_3(\mathtt{P}, \mathtt{cl}'', \chi_1(\iota), \mathtt{m}) = \\ \qquad\qquad \{\ \mathtt{t\ m(t'\ x)}\ \{\ \mathtt{e}\ \}\ \} \\ \mathcal{M}^{\mathtt{orig}}(\mathtt{P}, \mathtt{cl}'', \mathtt{m}) = \mathtt{cl}' \\ \sigma' = (\sigma(\mathtt{this}), \mathtt{v_1}, \mathtt{cl}') \\ \mathtt{e},\ \sigma',\ \chi_1 \rightsquigarrow_{\mathtt{P}} \mathtt{v}, \chi'\end{array}}{\mathtt{super.m(e_a)},\ \sigma,\ \chi \rightsquigarrow_{\mathtt{P}} \mathtt{v}, \chi'}$$

5.3 Type System

The judgment $\mathtt{P} \vdash \mathtt{tr}' \lesssim \mathtt{tr}$ says that trait $\mathtt{tr}'$ may replace another trait $\mathtt{tr}$. It requires that $\mathtt{tr}'$ provides all the methods that $\mathtt{tr}$ does (with the same signatures, but possibly different bodies), and that any methods provided or required by $\mathtt{tr}'$ are also provided or required in $\mathtt{tr}$.

$$\frac{\begin{array}{l}\mathtt{P} \vdash \mathtt{tr} \diamond_{\mathrm{trait}} \qquad\qquad\qquad \mathtt{P} \vdash \mathtt{tr}' \diamond_{\mathrm{trait}} \\ \forall \mathtt{m}:\ \ \mathtt{t_0\ m(t_1\ x)}\{\dots\} \in \mathtt{P}^{\mathtt{mth}}(\mathtt{tr}, \mathtt{m}) \Longrightarrow \mathtt{t_0\ m(t_1\ x)}\{\dots\} \in \mathtt{P}^{\mathtt{mth}}(\mathtt{tr}', \mathtt{m}) \\ \forall \mathtt{m}:\ \ \mathcal{MS}\mathrm{ig}_2(\mathtt{P}, \mathtt{tr}', \mathtt{m}) \subseteq \mathcal{MS}\mathrm{ig}_2(\mathtt{P}, \mathtt{tr}, \mathtt{m})\end{array}}{\mathtt{P} \vdash \mathtt{tr}' \lesssim \mathtt{tr}}$$

We require $\mathcal{MS}\mathrm{ig}_2(\mathtt{P}, \mathtt{tr}', \mathtt{m}) \subseteq \mathcal{MS}\mathrm{ig}_2(\mathtt{P}, \mathtt{tr}, \mathtt{m})$[8] because $\mathtt{P} \vdash \mathtt{tr}' \lesssim \mathtt{tr}$ and $\mathtt{P},\ \mathtt{t'\ x}, \mathtt{tr'\ this} \vdash_3 \mathtt{e} : \mathtt{t}$ should imply $\mathtt{P}, \mathtt{t'\ x}, \mathtt{tr\ this} \vdash_3 \mathtt{e} : \mathtt{t}$ – namely, if an object contains a trait placeholder $\mathtt{tr}$, which is replaced by $\mathtt{tr}'$, then it may execute method body $\mathtt{e}$ which was defined in $\mathtt{tr}'$. To satisfy $\mathtt{P},\ \mathtt{t'\ x}, \mathtt{tr\ this} \vdash_3 \mathtt{e} : \mathtt{t}$ for the case where $\mathtt{e{=}this}$, we need $\mathtt{P} \vdash_3 \mathtt{tr} \leq \mathtt{tr}'$, which requires $\mathcal{MS}\mathrm{ig}_2(\mathtt{P}, \mathtt{tr}', \mathtt{m}) \subseteq \mathcal{MS}\mathrm{ig}_2(\mathtt{P}, \mathtt{tr}, \mathtt{m})$.

In our example, $\_ \vdash \mathtt{TrtB2} \lesssim \mathtt{TrtB}$, and $\_ \not\vdash \mathtt{TrtB} \lesssim \mathtt{TrtB2}$ – because TrtB2 has a method body for m1, and TrtB has not.

Because trait substitutability implies subtypes, in *Chai*₃ we extend the subtype relationship from figure 7 as follows:

[8] Andrew Black suggested to us that we could weaken our original requirement of $\mathcal{MS}\mathrm{ig}_2(\mathtt{P}, \mathtt{tr}', \mathtt{m}) = \mathcal{MS}\mathrm{ig}_2(\mathtt{P}, \mathtt{tr}, \mathtt{m})$.

$$\frac{\mathtt{P} \vdash_2 \mathtt{t}' \leq \mathtt{t}}{\mathtt{P} \vdash_3 \mathtt{t}' \leq \mathtt{t}} \qquad \frac{\mathtt{P} \vdash \mathtt{tr}' \lesssim \mathtt{tr}}{\mathtt{P} \vdash_3 \mathtt{tr} \leq \mathtt{tr}'} \qquad \frac{\mathtt{P} \vdash_3 \mathtt{t}' \leq \mathtt{t}'' \quad \text{and} \quad \mathtt{P} \vdash_3 \mathtt{t}'' \leq \mathtt{t}}{\mathtt{P} \vdash_3 \mathtt{t}' \leq \mathtt{t}}$$

The type system of *Chai*3 is identical to that of *Chai*2, except for the new definition of subtypes ($\mathtt{P} \vdash_3 \mathtt{t}' \leq \mathtt{t}$) and complete classes ($\mathtt{P} \vdash_3 \mathtt{cl} \diamond_{\text{cmpl}}$), and the addition of the rule for mutation expressions. It requires that the type of `e` should be any class or trait `t`, that `t` should be using a trait `tr`, and that `tr′` may replace `tr`in `t`. Then, the substitution of `tr` through `tr′` in `e` has type `t`:

mutate

$$\frac{\begin{array}{l}\mathtt{P}, \Gamma \vdash_3 \mathtt{e} : \mathtt{t} \\ \mathtt{tr} \in \mathcal{U}se^*(\mathtt{P}, \mathtt{t}) \\ \mathtt{P} \vdash \mathtt{tr}' \lesssim \mathtt{tr}\end{array}}{\mathtt{P}, \Gamma \vdash_3 \mathtt{e} < \mathtt{tr} \mapsto \mathtt{tr}' > : \mathtt{t}}$$

5.4 Type Soundness

Agreement for *Chai*3 is defined in the following. In addition to the properties for agreement in *Chai*2, for *Chai*3 we use the new subtype relation ($\mathtt{P} \vdash_3 \mathtt{cl} \leq \mathtt{t}$), and require that all traits used by class `cl` should appear in the representation of the objects, and that all traits have been replaced by substitutable traits:

$$\frac{\begin{array}{l}\chi(\iota) = [\![\ \mathtt{cl} \ \| \ \ldots \ \| \ \mathtt{tr}_1 : \mathtt{tr}'_1, \ldots, \mathtt{tr}_n : \mathtt{tr}'_n \ ]\!] \\ \{ \ \mathtt{tr}_1 \ldots \mathtt{tr}_n \ \} = \mathcal{U}se^*(\mathtt{P}, \mathtt{cl}) \\ \forall \mathtt{i} \in 1, \ldots, \mathtt{n}: \quad \mathtt{P} \vdash \mathtt{tr}'_i \lesssim \mathtt{tr}_i \\ \mathtt{P} \vdash_3 \mathtt{cl} \leq \mathtt{t} \\ \mathcal{F}(\mathtt{P}, \mathtt{cl}, \mathtt{f}) = \mathtt{t}' \implies \mathtt{P}, \chi \vdash_3 \chi(\iota)(\mathtt{f}) \lhd \mathtt{t}'\end{array}}{\mathtt{P}, \chi \vdash_3 \iota \lhd \mathtt{t}}$$

The counterparts to the properties from lemmas 1 and 3 hold for *Chai*3.

Lemma 5. *For program* `P` *with* $\vdash_3 \mathtt{P}$*, classes* `cl`*,* `cl′`*, types* `t`*,* `t′`*,* `t″`*, with* $\mathtt{P} \vdash_3 \mathtt{cl} \leq \mathtt{cl}'$*, and* $\mathtt{P} \vdash_3 \mathtt{t} \leq \mathtt{t}'$:

1. $\mathcal{F}(\mathtt{P}, \mathtt{cl}', \mathtt{f}) = \mathtt{t} \implies \mathcal{F}(\mathtt{P}, \mathtt{cl}, \mathtt{f}) = \mathtt{t}$.
2. $\mathcal{MS}\text{ig}_2(\mathtt{P}, \mathtt{t}', \mathtt{m}) \subseteq \mathcal{MS}\text{ig}_2(\mathtt{P}, \mathtt{t}, \mathtt{m})$.
3. $\mathtt{P}, \mathtt{t_a}\ \mathtt{x},\ \mathtt{t}'\ \mathtt{this} \vdash_3 \mathtt{e} : \mathtt{t}'' \implies \mathtt{P}, \mathtt{t_a}\ \mathtt{x},\ \mathtt{t}\ \mathtt{this} \vdash_3 \mathtt{e} : \mathtt{t}''$.
4. $\mathtt{P}, \sigma \vdash_3 \iota \lhd \mathtt{cl}$, *and* $\mathtt{P} \vdash_3 \mathtt{cl} \diamond_{\text{cmpl}}$, *and* $\mathcal{M}^{\text{orig}}(\mathtt{P}, \mathtt{cl}', \mathtt{m}) = \{ \ \mathtt{cl}'' \ \}$, *and* $\mathtt{t}_0\ \mathtt{m}(\mathtt{t}_1\ \mathtt{x})\{\mathtt{e}\} \in \mathcal{M}_3(\mathtt{P}, \mathtt{cl}, \chi(\iota), \mathtt{m})$, $\implies$
   – $\mathtt{P} \vdash_3 \mathtt{cl}' \leq \mathtt{cl}''$
   – $\mathtt{P}, \mathtt{t}_1\ \mathtt{x},\ \mathtt{cl}''\ \mathtt{this} \vdash_3 \mathtt{e}'' : \mathtt{t}_0$.
5. $\mathtt{t}_0\ \mathtt{m}(\mathtt{t}_1\ \mathtt{x}) \in \mathcal{MS}\text{ig}_2(\mathtt{P}, \mathtt{t}, \mathtt{m})$, *and* $\mathtt{P}, \sigma \vdash_3 \iota \lhd \mathtt{cl}$, *and* $\mathtt{P} \vdash_3 \mathtt{cl}' \leq \mathtt{t}$, *and* $\mathtt{P} \vdash_3 \mathtt{cl} \diamond_{\text{cmpl}} \implies \mathcal{M}_3(\mathtt{P}, \mathtt{cl}', \chi(\iota), \mathtt{m}) = \{ \ \mathtt{t}_0\ \mathtt{m}(\mathtt{t}_1\ \mathtt{x})\{ \ldots \} \ \}$.

We can now prove soundness for the type system of *Chai*$_3$:

Theorem 6 (Type Soundness of *Chai*$_3$**).** *For any program* $\mathtt{P}$*, environment* Γ*, expression* $\mathtt{e}$*, stack* σ*, heap* χ*, type* $\mathtt{t}$*, where* $\vdash_3 \mathtt{P}$*, and* $\mathtt{P}, \Gamma \vdash_3 \mathtt{e} : \mathtt{t}$ *and* $\mathtt{P}, \Gamma \vdash_3 \sigma, \chi$ *and* $\mathtt{e}, \sigma, \chi \leadsto_{\mathtt{P}} \mathtt{r}, \chi'$*:*

$$\mathtt{P}, \chi' \vdash \mathtt{r} \lhd \mathtt{t} \quad \textit{or} \quad \mathtt{r} = \mathtt{nullPointerExc} \qquad \textit{and} \qquad \mathtt{P}, \Gamma \vdash_3 \sigma, \chi'.$$

6 Implementation

This section describes the translation of a program in *Chai* (the source language) to one in Java (the target language). This is implemented by a mapping from traits and classes in *Chai* to entities in Java[9] There are several possible mappings we could have chosen for this purpose; we could map a class (and all the behaviour it includes from traits) in *Chai* to a single class in Java. Instead, we choose a slightly more complex mapping, which represents traits in Java by classes which are instantiated to give *proxy objects* to which behaviour can be delegated by a class which uses those traits. This allows us to implement the dynamic trait substitution of *Chai*$_3$.

Every trait `tr` is represented by an object of type `tr_impl`, and contains a field called `user_proxy` of type `tr_user`. The `user_proxy` field always stores a reference to an object of the trait or class that uses this trait. Also, for any class or trait, there are fields `tr′_proxy` for all traits `tr′` used by the class or trait. Each of these is a reference to an object of the relevant type `tr′_interface`

Take, for example, a class `D` which uses a trait `T3`, and `T3` uses traits `T1` and `T2`. Because $\mathtt{T3} \in \mathtt{P}^{\mathtt{use}}(\mathtt{D})$, the `D` object contains a reference to a `T3_impl` object. Similarly because $\mathtt{T1} \in \mathtt{P}^{\mathtt{use}}(\mathtt{T3})$ and $\mathtt{T2} \in \mathtt{P}^{\mathtt{use}}(\mathtt{T3})$, the `T3` object contains references to `T1_impl` and `T2_impl` objects.

In order for this arrangement to be type correct, all classes `tr_impl` must implement `tr_interface`, and also `tr′_user` for all `tr′` such that $\mathtt{tr}' \in \mathtt{P}^{\mathtt{use}}(\mathtt{tr})$. Additionally, classes that use traits must implement the appropriate `tr_user` interfaces (in the example, $\mathtt{T3} \in \mathtt{P}^{\mathtt{use}}(\mathtt{D})$ and so `D` must implement `T3_user`).

The reason that the type of the fields `tr_proxy` is `tr_interface` (and not `tr_impl`) is to allow different values stored in the field to refer to different trait implementation objects (provided that they implement `tr_interface`), and support trait substitutions (under the restrictions described by *Chai*$_3$).

In more detail, every trait `tr` in *Chai* is mapped to three entities in Java:

1. A *trait interface* containing all the provided methods of `tr`.
2. A *trait-user interface* containing all the required method signatures (i.e. those expected to be provided by the user of the trait `tr`), as well as all the provided method signatures of `tr` (see below).
3. A *trait implementation class* which contains the definitions for the provided methods of the trait, proxy fields for the user of the trait and all used traits,

[9] Similarly, Java-mixins were implemented through a mapping from Jam into Java[2].

as well as delegation method stubs for acquired methods, which forward method calls to the used trait proxy objects.

A class in *Chai* is mapped to a class in Java, with the addition of proxy fields for used traits, implements declarations for the trait-user interfaces of traits used by the class, and method stubs for acquired methods and superclass methods required by a used trait.

To preserve the intended semantics of the flattening property (see section 3.3), it is necessary that the use of the expression `this` within a trait proxy is translated to refer the object belonging to the class which uses the trait (note that there may be several levels of intervening trait proxies between the trait proxy and this object). The reason that this is necessary, is that declarations of methods "most local" to the eventual user of a trait have precedence, therefore to preserve the flattening property, we must start the search for a method implementation from this user object itself and work upward into traits represented by proxy objects.

Prototype. The prototype implementation of the compiler is written in Java. At present it supports all of the features of $Chai_1$, and would easily accommodate extensions to support $Chai_2$ and $Chai_3$. The compiler, including full source code, is available from `http://chai-t.sourceforge.net/`.

7 Conclusions, Related and Further Work

We have developed three extensions to a minimal Java-like language incorporating traits, have proven soundness of the type systems, and have outlined our prototype implementation.

The main issues we had to address during the design of *Chai* were:

– The precise semantics of using a trait as part of a class in Java;
– How to perform type-checking on traits, and in particular how to avoid having to type-check the same method body in each class that uses a trait;
– The reflection of calls to `super` in the requirements part of traits
– In how far classes have to be complete, *i.e.,* provide method bodies for all the methods required by the traits they are using;
– Subtype relationships between classes and traits, as required in $Chai_2$; interestingly, a trait may require *more* methods than a supertype trait;
– Dynamic substitution of traits, and the semantics of method lookup in $Chai_3$;
– The trait substitutability relationship in $Chai_3$; interestingly, substitutability in $Chai_3$ does not imply subtype in $Chai_2$.

Recently, and especially after the application of traits to Smalltalk [18, 19], the interest in traits has boomed. In [10] a imperative calculus for traits in the language Moby is developed. The acquisition of methods trough the use of traits is modeled through "class evaluation" which returns flattened classes. As in our work, alias and exclusion of methods in [10] is accompanied by method signatures; unlike our work, traits in [10] may require the presence of fields.

In FTJ [13] traits are added to Featherweight Java[12]; the system is functional, and traits are treated as a class creation mechanism, similar to $Chai_1$. The full calculus of FTJ and a proof of soundness of the type system is presented.

Traits are part of the language Scala [15], where they play similar rôle to that of $Chai_1$ and $Chai_2$. Scala incorporates many advanced features *e.g.,* generics, and dependent types; it is unknown whether its type system is decidable [16].

The Software Composition group at the University of Berne [8] contains a large center for the research around the design, semantics, and application of traits. Tools for Traits for Squeak are being developed, and Microsoft research is sponsoring the design and implementation of traits for C#.

In further work, we would like to refine our model to support overloading. We also want to revisit and reconsider the design decisions in $Chai_2$ and $Chai_3$; so far they were taken just with the aim to obtain type soundness, but we should explore their implications for the style of programming. We also want to explore the design space for traits, its relation with generic features [6], possibly also incorporate polymorphic features into traits. We also would like to consider generalization of the languages, *e.g.,* allow classes to have trait glue, or allow trait glue to require fields.

Acknowledgments. We are deeply grateful to Andrew Black, Tim Sheard, and their research group, for insightful suggestions, pertinent questions, and detailed comments on an earlier version of this work. Their feedback has greatly improved the suggested language features in $Chai_1$, $Chai_2$ and $Chai_3$, the explanations, and presentation – we have not yet had the time to incorporate all of it. Chuan-Kai Lin and Andrew Black suggested the correct fomalization of `super`.

The anonymous ECOOP referees, Alex Buckley, Christopher Anderson, Nick Cameron, and Rok Strnisa gave us valuable feedback. Nobuko Yoshida suggested the distillation of the essential properties used in the proof of soundness for $Chai_1$, $Chai_2$ and $Chai_3$, as in lemmas 1, 3 and 5.

References

1. Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hölzle, John Maloney, Randall B. Smith, and David Ungar. *The Self 4.0 Programmer's Reference Manual.* Sun Microsystems, Inc., 1995.
2. Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam - A Smooth Extension of Java with Mixins. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 154–178. Springer-Verlag, 2000.
3. Andrew Black, Nathanael Schärli, and Stéphane Ducasse. Applying Traits to the Smalltalk Collection Hierarchy. pages 47–64. ACM Conference on Object Oriented Systems, Languages and Applications (OOPSLA), October 2003.

4. Gilad Bracha. *The programming language jigsaw: mixins, modularity and multiple inheritance.* PhD thesis, University of Utah, 1992.
5. Gilad Bracha and William Cook. Mixin-Based Inheritance. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, 1990. ACM Press.
6. Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.
7. S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic object re-classification. In *ECOOP'01*, LNCS 2072, pages 130–149. Springer, 2001.
8. Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, and Roel Wuyts. Traits - Composable Units of Behaviour. University of Berne, Software Composition Group, `http://www.iam.unibe.ch/ scg/Research/Traits/index.html`.
9. Erik Ernst. Family Polymorphism. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 303–326. Springer-Verlag, 2001.
10. Kathleen Fisher and John Reppy. Statically Typed Traits. Technical Report TR-2003-13, Department of Computer Science, University of Chicago, December 2003. presented at FOOL, January 2004.
11. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and Mixins. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 171–183, New York, NY, 1998.
12. Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.
13. L. Liquori and A.Spiwack. Featherweight-Trait Java, A Trait-based Extension for FJ. 2004, http://www-sop.inria.fr/mirho/Luigi.Liquori/PAPERS/ftj.ps.gz.
14. Bertrand Meyer. *Eiffel: the Language.* Prentice-Hall, 1988.
15. Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenge. The Scala Language Specification Version 1.0. Technical report, Programming Methods Laboratory, EPFL, Switzerland, 2004.
16. Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A Nominal Theory of Objects with Dependent Types. In *Proc. ECOOP'03*, Springer LNCS, 2003.
17. Philip J. Quitslund and Andrew P. Black. Java with Traits — Improving Opportunities for Reuse. In *The MASPEGHI Workshop at ECOOP 2004.*
18. Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable Units of Behavior. European Conference on Object-Oriented Programming (ECOOP), Springer LNCS 2743, July 2003.

19. Nathanael Schärli, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, and Andrew Black. Traits: The Formal Model. Technical Report IAM-02-006, Institut für Informatik, Universität Bern, Switzerland, November 2002.
20. Charles Smith. Typed Traits, September. MSc thesis - Department of Computing, Imperial College London, September 2004, `http://chai-t.sourceforge.net/`.
21. B. Stroustrup. Multiple inheritance for C++. In *Proceedings of the Spring 1987 European Unix Users Group Conference*, Helsinki, 1987.
22. David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242. ACM Press, 1987.

A Type System for Reachability and Acyclicity

Yi Lu and John Potter

Programming Languages and Compilers Group,
School of Computer Science and Engineering,
The University of New South Wales,
Sydney 2052, Australia
{ylu, potter}@cse.unsw.edu.au

Abstract. The desire for compile-time knowledge about the structure of heap contexts is currently increasing in many areas. However, approaches using whole program analysis are too weak in terms of both efficiency and accuracy. This paper presents a novel type system that enforces programmer-defined constraints on reachability via references or pointers, and restricts reference cycles to be within definable parts of the heap. Such constraints can be useful for program understanding and reasoning about effects and invariants, for information flow security, and for run-time optimizations and memory management.

1 Introduction

Pointers and references allow run-time sharing of data structures. In most software, references are unavoidable for pragmatic efficiency reasons, even though they complicate program reasoning and their usage is error-prone. When data structures are mutable, problems are inevitable. Reference cycles, either direct or indirect, can cause serious programming errors; a sequence of method calls following a reference cycle, may unexpectedly break invariants of local states or cause non-termination. Reference cycles also complicate the task of memory management and cloning; for instance, safety of explicit memory deallocation may be difficult in the presence of arbitrary cycles and automatic garbage collection cannot rely on reference counting alone in the presence of cycles.

Object-oriented programming languages use reference semantics for objects which, together with subtyping and a generic coding style, increase the likelihood of unintended object reference cycles. Common object-oriented design patterns, like the decorator, often require an absence of such cycles. Wrapping an object with self-cycles yields a design problem: should the self-cycles be re-routed via the wrapper or not? When is it safe to wrap an object?

The potential for cycles limits our ability to use class invariants in reasoning about the state of an object before and after method calls. If a method indirectly calls back on an object, via an indirect reference cycle, say, then the call-back may be entering the object in an invalid state, so the call-back code may be working outside of its assumed precondition, and furthermore the original call on the object may not be aware of the indirect effect of the call-back, so its desired postcondition may not be met. With reference cycles we cannot presume that an invariant will hold for all calls on an object. This problem manifests itself in a language like Eiffel which allows run-time assertion checking for class invariants – when should the invariant hold? The problem has been highlighted in recent work on verification of object-oriented programs [3, 21] which suggests enriching the program state to track when object invariants hold; they incorporate a notion of dynamic ownership.

Reference problems are difficult to reason about because in most cases callbacks and infinite loops may be caused by indirect references which a programmer may be unaware of. Shape analysis attempts to characterize the shape of data structures via whole program analysis [26]. However, these approaches suffer from exponential complexity and cannot be accurate especially in the present of cyclic reference structures. They are also hard to scale to large or incomplete programs because of lack of modularity in the analysis techniques.

In this paper, we employ a type system, called *Acyclic Region Type System*, that allows programmers to specify desired reachability relations between objects via regions. We use the name *ARTS* to refer to our type system, the overall model, and underlying language. Regions partition the heap into distinct logical blocks of memory, and every object lives in a fixed region determined from its type. Object reference cycles are only allowed within the same region, that is, regions are partially ordered by the object reachability relation. In this way, programmers are able to express where cycles are allowed, by creating objects in the same region; they can also forbid unwanted cycles by using different regions. ARTS allows modular reasoning on a unbounded number of regions and separate compilation is possible. We also present a dynamic semantics that allows us to formalize the key structural invariant: object references respect region reachability, so that object cycles occur only within regions. Besides program reasoning, our type system has potential application in many areas: information flow security, memory management, data copying or cloning, deadlock avoidance, and shape analysis where cycles remain an obstacle.

This paper is organized as follows: an informal overview of ARTS is given in the next section with some program examples. The core language of ARTS is formalized in Section 3, where along we present static semantics. Dynamic semantics and some important properties of the type system are given in Section 4. Discussion and related work are given in Section 5 and Section 6. Section 7 briefly concludes the paper along with some thoughts on future directions.

2 Overview of the Acyclic Region Type System

ARTS uses region-based types to capture the potential of an object to reach other objects, directly or indirectly. In typed languages, the occurrence of a cycle in the run-time object graph implies there must be a cycle in the type dependency graph; in other words, if there is no cycle in the type dependency graph, then there will be none in the run-time object graph. If there are type-level cycles, the type system is powerless to prevent cyclic references, even if they are undesirable.

With our acyclic region type system, we can enforce one-way reachability for objects, that is, one object may reach another via a path that is not part of a cycle in the object graph. *Acyclic reachability* for regions is the key concept in our model: it is a strict partial order which we denote by $\triangleright$. Underlying the design of ARTS is the acyclic graph induced by the partition of a directed graph into its strongly connected components (*scc*s). All cycles within the original graph must occur within these components. We have designed our type system so that regions and acyclic reachability provide a static abstraction of the strongly connected components of the dynamic object graph and object reachability between them. The key idea is to formulate static constraints on the object graph by specifying regions within which all reference cycles are trapped. Regions are disjoint sets of objects and every object lives in the same region for its lifetime. We impose constraints on region reachability, and guarantee that inter-object references respect these inter-regional reachability constraints.

To use such a model, programmers must be able to decide when they want to ban the possibility of cycles or aliases emanating from particular object fields. This will be clearer if we look at an example illustrating a simple use of ARTS.

```
class A<p>
  r from p;
  B<r> f;
  ...
    // assert a property about this object's fields
    f.m();
    // assert the same property
```

This code shows the simplest case of ARTS. Class `A` is parameterized with a region parameter `p`, which represents the region the current object lives in. A region `r` is defined within the class with the constraint `r from p`, which means `p` one-way reaches `r`, or `p` $\triangleright$ `r`. This implies there may be a reference sequence starting in `p` that ends in `r` but not vice versa. The type of the field `f` is `B<r>` which means `f` references an object living in the region `r`. In this case, the object referenced by `f` can never hold a reference to the current object because of the order we force on their types. So any method call made on `f` cannot reenter the current object. Such knowledge allows us to assert properties that are necessarily invariant during the call, such as the reference stored in the field `f`.

ARTS can significantly improve program understanding. Programmers are able to express reference reachability between objects via types, and know that two references cannot be direct aliases if their objects live in different regions. The next example will show how to express complex data structures, such as linked lists with iterators, in ARTS.

2.1 A Linked List Example

Linked lists provide a common example for demonstrating expressiveness of language features dealing with references. Our list example will show how regions work and how acyclic properties are expressed in a program. In particular, this example shows how the list data structure is handled so that a list object can never reach itself via its data objects. In other words, the data objects contained by a list must not reach and thereby alter the list, which could cause iterators on the list to fail, for example. The data objects themselves may well be shared by other parts of the program.

```
class List<list, data from list>
  link from list to data;
  Link<link, data> head;
  Link<link, data> tail;
  void addElement(Data<data> d)
    head = new Link<link, data>(head, d);
    if (tail == null) tail = head;
    tail.next = head;
  Iterator<list, data> getIterator()
    return new Iterator<list, data>(head);

class Link<link, data from link>
  Link<link, data> next;
  Data<data> d;
  Link(Link<link, data> next, Data<data> d)
    this.next = next;
    this.d = d;

class Iterator<list, data from list>
  Link<List<list, data>.link, data> current;
  Iterator(Link<List<list, data>.link, data> current)
    this.current = current;
  void next()
    current = current.next;
  Data<data> element()
    return current.data;
```

A list object is implemented by a cycle of link objects. Iterator objects are created inside the list's region, and are used to access the data stored in link objects. They can name the `link` region through a qualification over the type of the list. This removes the naming restriction in instance-based parametric type systems such as ownership types (see Section 6).

The relations between the regions in the example are shown in Figure 1. All link objects live in the same region defined in the `List` class, and all list, link and iterator objects have access to the data objects in another region given by a parameter of their classes. The type system enforces a lack of cycles between regions. Data objects can never reference the list or iterator objects; these are shown in the graph as 'bad references'. As expected, cyclic references are allowed within a region. In the example, link objects form a cycle within their region.

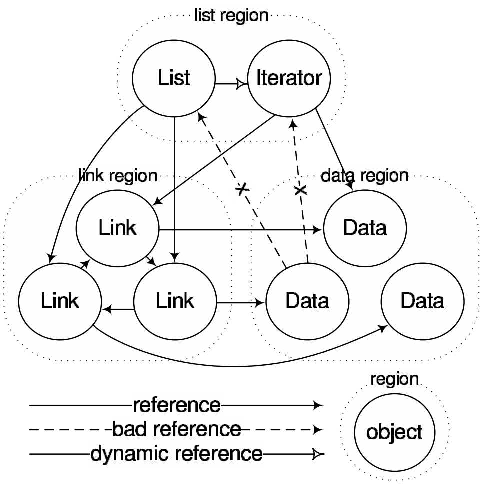


Fig. 1. List Example

3 The Language and Static Semantics of ARTS

In this section we formalize our model, providing an abstract syntax which amounts to a simple Java-like language, and a type system that captures the desired properties. Both the language and the type system are described in some detail, and we illustrate particular language features with small examples.

The key technical contribution of the paper is the way in which new region definitions define a refinement of the acyclic reachability ordering on existing regions. How does this work? Consider instead how subclass definition extends the acyclic inheritance relation on existing classes: the key ensuring that inheritance is acyclic is to ensure that there is an order of definition of classes in which subclasses are defined after superclasses. For region definitions in ARTS, we also rely on a definition order to ensure acyclicity. But extra care is needed. New regions not only extend the reachability, but also refine it by placing new regions between existing ones. The real trick to make the system work, is that, in the case of region refinement, the reachability between the existing regions must already be derivable before the new region is introduced.

3.1 An Abstract Syntax for ARTS

To simplify the abstract syntax presented in Table 1, we use a few abbreviations. The overbar is used for a sequence of constructs; for example, $\overline{\sigma}$ is used for a possibly empty sequence σ_1 ... σ_n, as are $\overline{p}$, $\overline{q}$, $\overline{fd}$, $\overline{mth}$ and $\overline{e}$. Similarly, $\overline{t\ x}$ stands for a possibly empty sequence of pairs $t_1\ x_1$ ... $t_n\ x_n$. In the class production, $[t]_{opt}$ is an optional part of the class. In the type system the equivalence symbol $\equiv$ denotes syntactic equivalence; it is used in defining syntactic lookup and substitution functions in Table 3. Just as in Java, `this` is a distinguished

Table 1. Abstract Syntax

$c \in$ ClassName; $r \in$ RegionName; $x \in$ VarName; $f \in$ FieldName; $m \in$ MethodName

$cls \in$ Class	$::= c\ \overline{p}\ [t]_{opt}\ \overline{q}\ \overline{fd}\ \overline{mth}$
$p, q \in$ RegionConstraint	$::= \overline{\sigma} \triangleright \sigma \triangleright \overline{\sigma}$
$t \in$ Type	$::= c\langle\overline{\sigma}\rangle$
$\sigma \in$ Region	$::= \texttt{base} \mid t.r \mid r$
$fd \in$ Field	$::= t\ f$
$mth \in$ Method	$::= t\ m(\overline{t\ x})\ e$
$e \in$ Expression	$::= x \mid \texttt{new}\ t \mid \texttt{null} \mid e.f \mid e.f = e \mid e.m(\overline{e}) \mid e; e \mid \texttt{if}\ e\ e\ e$

variable name used to reference the target object for the current call, that is, the *current object*. In the concrete syntax we use in our examples, we use keywords such as `class` and `extends` for ease of reading.

Classes and Constrained Formal Parameters. Our syntax is close to Java except that classes are parameterized with region parameters and region names are defined as members within classes.

The relational symbol $\triangleright$, in region constraints, denotes the *acyclic reachability* relation between regions. In the concrete syntax, the region constraints $q \equiv \overline{\sigma} \triangleright r \triangleright \overline{\sigma'}$ that defines new region r is written as:

$$r\ \texttt{from}\ \sigma_1, ..., \sigma_m\ \texttt{to}\ \sigma'_1, ..., \sigma'_n \quad \text{where} \quad |\overline{\sigma}| = m \quad \text{and} \quad |\overline{\sigma'}| = n$$

The same applies to the constraints for the formal parameters $p \equiv \overline{\sigma} \triangleright r \triangleright \overline{\sigma'}$. The formal region parameters of a class are assumed to satisfy the reachability constraints specified in the `from` and `to` clauses of each parameter. They are used within a class to identify regions for objects used by the class. The first formal parameter denotes the region where the current object (`this`) lives. Our syntax for region names r does not distinguish between names of region parameters and locally defined regions. The reason for using two key words `from` and `to` in the concrete syntax instead of one is to make the constraints clear – the parameter or new region to be introduced occurs first.

Types. Classes are type schemas. A type is formed by binding the region parameters to actual regions in the environment where the type is formed. A type consists of a class name and the region arguments required by its class definition. The first region argument is the region where the object of this type resides. Unsurprisingly, for a type to be valid, the actual region arguments must satisfy the class constraints defined on the formal parameters.

Regions and Region Definitions. In ARTS, every object belongs to a fixed region for its entire lifetime. Region expressions are formed from the special global region `base`, from type-qualified regions, and region parameters. To ensure the acyclic property for the region reachability relation, the order of introduction of new regions in definitions is important. A region definition q introduces a fresh name for its region, together with constraints that specify its reachability

Table 2. Static Semantics Given Class Definitions Π

Well-Formed Program and Well-Ordered Class Definitions $\quad \vdash_P e;\ \ \vdash_c \Gamma;\ \ \Gamma \vdash_c cls$

$$\text{[PROGRAM]}\ \frac{\begin{array}{c}\vdash_c \Pi\\ \Pi \vdash_e e : t\end{array}}{\vdash_P \Pi\ e}\qquad \text{[CLS-DEF0]}\ \frac{}{\vdash_c \emptyset}\qquad \text{[CLS-DEFS]}\ \frac{\begin{array}{c}\vdash_c \Gamma\\ \Gamma \vdash_c cls\end{array}}{\vdash_c \Gamma, cls}$$

$$\text{[CLS-DEF]}\ \frac{\begin{array}{c} c \notin \Gamma \quad \overline{r} \equiv \mathcal{R}(\overline{p}) \quad E \equiv \Pi, \overline{p}, \texttt{this} : c\langle \overline{r} \rangle \\ \Pi \vdash_r \overline{p} \quad \Gamma, \overline{p} \vdash_r \overline{q} \quad E \vdash_f \overline{fd} \quad E \vdash_m \overline{mth} \\ [t \equiv c'\langle r_1, ...\rangle \ \ E \vdash_t t \ \ c' \in \Gamma]_{opt} \end{array}}{\Gamma \vdash_c c\ \overline{p}\ [t]_{opt}\ \overline{q}\ \overline{fd}\ \overline{mth}}$$

Well-Ordered Region Definitions $\quad E \vdash_r \overline{p};\ \ E \vdash_r p$

$$\text{[REG-DEF0]}\ \frac{}{E \vdash_r \emptyset}\qquad \text{[REG-DEFS]}\ \frac{E \vdash_r \overline{p} \quad E, \overline{p} \vdash_r p}{E \vdash_r \overline{p}, p}\qquad \text{[REG-DEF]}\ \frac{r \notin \mathcal{R}(E) \quad E \vdash_\sigma \overline{\sigma}, \overline{\sigma'} \quad E \vdash_\triangleright \overline{\sigma} \triangleright \overline{\sigma'}}{E \vdash_r \overline{\sigma} \triangleright r \triangleright \overline{\sigma'}}$$

Well-Defined Field and Method $\quad E \vdash_f fd;\ \ E \vdash_m mth$

$$\text{[FIELD]}\ \frac{\begin{array}{c} t_o \equiv c_o\langle r, ...\rangle \quad t \equiv c\langle \sigma, ...\rangle \\ E \vdash_e \texttt{this} : t_o \quad E \vdash_t t \quad E \vdash_\triangleright r \trianglerighteq \sigma \end{array}}{E \vdash_f t\ f}\qquad \text{[METHOD]}\ \frac{\begin{array}{c} E \vdash_t t, \overline{t} \\ E, \overline{x : t} \vdash_e e : t \end{array}}{E \vdash_m t\ m(\overline{t\ x})\ e}$$

Well-Formed Region and Type $\quad E \vdash_\sigma \sigma;\ \ E \vdash_t t$

$$\text{[REG-BASE]}\ \frac{}{E \vdash_\sigma \texttt{base}}\qquad \text{[REG-NAME]}\ \frac{r \in \mathcal{R}(E)}{E \vdash_\sigma r}\qquad \text{[REG-QUAL]}\ \frac{E \vdash_t t \quad t.r \in \mathcal{R}(t)}{E \vdash_\sigma t.r}\qquad \text{[TYPE]}\ \frac{\begin{array}{c} t \equiv c\langle \overline{\sigma} \rangle \quad \mathcal{C}(t) \equiv c\ \overline{p}\ ... \\ c \in E \quad E \vdash_\sigma \overline{\sigma} \quad E \vdash_\triangleright \overline{p} \end{array}}{E \vdash_t t}$$

Subtype $\quad \vdash_{<:} t <: t'$

$$\text{[SUBTYPE-REFL]}\ \frac{}{\vdash_{<:} t <: t}\qquad \text{[SUBTYPE-EXTEND]}\ \frac{\mathcal{C}(t) \equiv ...\ t'\ ...}{\vdash_{<:} t <: t'}\qquad \text{[SUBTYPE-TRANS]}\ \frac{\vdash_{<:} t <: t' \quad \vdash_{<:} t' <: t''}{\vdash_{<:} t <: t''}$$

Region Reachability $\quad E \vdash_\triangleright \sigma \triangleright \sigma';\ \ E \vdash_\triangleright \sigma \triangleright \sigma' \triangleright \sigma'';\ \ E \vdash_\triangleright \sigma \trianglerighteq \sigma'$

$$\text{[REACH-ENV]}\ \frac{\sigma \triangleright \sigma' \in E}{E \vdash_\triangleright \sigma \triangleright \sigma'}\qquad \text{[REACH-DEF-TO]}\ \frac{t.r \triangleright \sigma \in \mathcal{Q}(t)}{E \vdash_\triangleright t.r \triangleright \sigma}\qquad \text{[REACH-DEF-FROM]}\ \frac{\sigma \triangleright t.r \in \mathcal{Q}(t)}{E \vdash_\triangleright \sigma \triangleright t.r}\qquad \text{[REACH-TRANS]}\ \frac{E \vdash_\triangleright \sigma \triangleright \sigma' \triangleright \sigma''}{E \vdash_\triangleright \sigma \triangleright \sigma''}$$

$$\text{[REACH-COMB]}\ \frac{E \vdash_\triangleright \sigma \triangleright \sigma' \quad E \vdash_\triangleright \sigma' \triangleright \sigma''}{E \vdash_\triangleright \sigma \triangleright \sigma' \triangleright \sigma''}\qquad \text{[REACH-REFL]}\ \frac{}{E \vdash_\triangleright \sigma \trianglerighteq \sigma}\qquad \text{[REACH-EXT]}\ \frac{E \vdash_\triangleright \sigma \triangleright \sigma'}{E \vdash_\triangleright \sigma \trianglerighteq \sigma'}$$

Well-Formed Expression $\quad E \vdash_e e : t$

$$\text{[EXPR-VAR]}\ \frac{}{E \vdash_e x : E(x)}\qquad \text{[EXPR-NEW]}\ \frac{E \vdash_t t}{E \vdash_e \texttt{new}\ t : t}\qquad \text{[EXPR-NULL]}\ \frac{E \vdash_t t}{E \vdash_e \texttt{null} : t}$$

$$\text{[EXPR-FIELD]}\ \frac{\begin{array}{c} (t\ f) \in \mathcal{F}(t_o) \\ E \vdash_e e : t_o \end{array}}{E \vdash_e e.f : t}\qquad \text{[EXPR-ASSIGN]}\ \frac{\begin{array}{c} (t\ f) \in \mathcal{F}(t_o) \\ E \vdash_e e : t_o \quad E \vdash_e e' : t \end{array}}{E \vdash_e e.f = e' : t}\qquad \text{[EXPR-CALL]}\ \frac{\begin{array}{c} \mathcal{M}(t_o, m) \equiv (\overline{t}, \_, t, \_) \\ E \vdash_e e : t_o \quad E \vdash_e \overline{e : t} \end{array}}{E \vdash_e e.m(\overline{e}) : t}$$

$$\text{[EXPR-SEQ]}\ \frac{\begin{array}{c} E \vdash_e e : t \\ E \vdash_e e' : t' \end{array}}{E \vdash_e e; e' : t'}\qquad \text{[EXPR-IF]}\ \frac{\begin{array}{c} E \vdash_e e : t \\ E \vdash_e e' : t' \quad E \vdash_e e'' : t' \end{array}}{E \vdash_e \texttt{if}\ e\ e'\ e'' : t'}\qquad \text{[EXPR-SUBSUM]}\ \frac{\begin{array}{c} E \vdash_e e : t \\ \vdash_{<:} t <: t' \end{array}}{E \vdash_e e : t'}$$

Table 3. Auxiliary Lookup Functions for Static Semantics

$$\frac{t \equiv c\langle\overline{\sigma}\rangle \quad \overline{r} \equiv \mathcal{R}(\overline{p}) \quad \overline{r'} \equiv \mathcal{R}(\overline{q}) \quad \Pi(c) \equiv cls \equiv c\ \overline{p}\ ...\overline{q}\ ...}{\mathcal{C}(t) \equiv cls[\overline{\sigma/r}, \overline{t.r'/r'}]}\ [\text{LOOKUP-CLASS}] \qquad \frac{\mathcal{C}(t) \equiv ...\ \overline{q}\ ...}{\mathcal{Q}(t) \equiv \overline{q}}\ [\text{LOOKUP-DEF}]$$

$$\frac{\mathcal{C}(t_o) \equiv ...\ [t']_{opt}\ ...\ \overline{t\ f}}{\mathcal{F}(t_o) \equiv [\mathcal{F}(t')]_{opt}, \overline{t\ f}}\ [\text{LOOKUP-FIELD}] \qquad \frac{\mathcal{C}(t_o) \equiv ...\ t\ m(\overline{t\ x})\ e\ ...}{\mathcal{M}(t_o, m) \equiv (\overline{t}, \overline{x}, t, e)}\ [\text{LOOKUP-METHOD}] \qquad \frac{\mathcal{C}(t_o) \equiv ...\ t\ ...\ \overline{mth}\ ... \quad m \notin \overline{mth}}{\mathcal{M}(t_o, m) \equiv \mathcal{M}(t, m)}\ [\text{LOOKUP-METHOD}']$$

$$\frac{}{\mathcal{R}(\emptyset) \equiv \emptyset \quad \mathcal{R}(E, cls) \equiv \mathcal{R}(E) \quad \mathcal{R}(E, p) \equiv \mathcal{R}(E), \mathcal{R}(p) \quad \mathcal{R}(E, x : t) \equiv \mathcal{R}(E)}\ [\text{LOOKUP-REGION-ENV}]$$

$$\frac{}{\mathcal{R}(\overline{\sigma} \triangleright \sigma'' \triangleright \overline{\sigma'}) \equiv \sigma'' \quad \mathcal{R}(\overline{q}, q) \equiv \mathcal{R}(\overline{q}), \mathcal{R}(q) \quad \mathcal{R}(t) \equiv \mathcal{R}(\mathcal{Q}(t))}\ [\text{LOOKUP-REGION-DEF}]$$

properties in terms of previously defined regions. Furthermore these constraints must not impose any further requirement on the reachability relation for the previously defined regions. In this way we are able to inhibit cycles in the region reachability relation. This is checked and enforced by the type system as formalized in Section 3.2.

Compared with constraints on formal parameters, which just impose requirements on the actual region arguments for a class, region definitions actually determine the region structure for the system. Every class defines a type schema, with its own locally defined regions. We allow types (not just class names) to qualify region names – every such qualified region $t.r$ uniquely determines a particular region in the system. To guarantee this uniqueness, we do not allow regions to be inherited by subclasses; otherwise the same region could be identified through a subtype which inherits it. In fact it is straightforward to allow region inheritance, but adds no extra expressivity, and slightly complicates the type system.

Interestingly, regions and types are recursively defined: regions are defined within classes and named via type qualification; types are formed by binding class parameters with actual regions. Because of this recursive structure there are an unbounded number of regions in the system, all globally accessible via region path expressions. Fortunately the programmer does not have to deal with global region names, because the region parameters of a class localize the expression of the regions relevant for a class, so normally region expressions do not need to be nested through more than one level of type qualification.

The primary goal of ARTS is to allow programmers to define a static preorder on the run-time object structure which induces a partial order on the scc's of the objects, thereby restricting where cycles can occur. By making class-defined region names publicly accessible we keep flexibility in that every ground region has a unique global name. Ground regions are those with no free region parameter: either `base`, or a region $t.r$ qualified by a ground type t; a ground type is one with no free region parameter. Objects of the same types (*not just the same class*) share the same region; objects of different type may occupy the same region, but their types must share the same first region argument because it determines the region an object lives in).

The special region `base` is pre-defined and is the only unqualified ground region in the system. An important note here is that regions do *not* have to be reachable from `base` or vice-versa; also any root object (objects created in the main method) does *not* have to be placed in `base`. `base` is just the base region used to name any other ground region. The ability to name a region is different from the ability to access a region, which is determined by the `to` and `from` clauses of the region definitions. In fact, we could allow the expression defining the main method of a program to define its own regions, making the use of global regions superfluous.

All region constraints are statically verified by the type system to ensure global acyclicity. Any strongly connected component in the object heap during the execution of an ARTS program must occur entirely within a region. Regions need not exist at run-time because the expression language only uses types for object allocation, and regions can be erased from that with no change in behaviour. Regions are only used in type checking to help organize and reason about cyclic references and object reachability.

Fields and Methods. In ARTS, the structural invariant is maintained by imposing a stronger restriction on object fields than on other reference-valued entities, namely method arguments and results. Object fields are singled out for special attention because they can form unwanted hard-to-detect cycles in object graphs. The structural invariant states that if an object contains a reference to another object then both objects must be in the same region or the region of the former object must be able to reach the region of the latter object with no return reference possible.

However, we do allow method arguments to access objects which are not accessible to the current object, that is, the regions of method parameters and results do not have to be reachable from the region of the `this` object. This implies that we still allow inter-region callbacks, but that they occur within method scope, and the method's type signature indicates whether any such call-back is allowed or not. These dynamic references can be considered safe because they emanate from the calling stack rather than the heap, which is not referenceable from the heap and deallocated at the exit of the method.

It is worth acknowledging at this stage that our choice of restrictions for field and method access is somewhat arbitrary. It is easy to modify the type rules to enforce different invariants; for example, if we wish to block the possibility of cycles with dynamic references as well, then we could impose the restriction that all method arguments and results are accessible from the region of `this` just like we do to fields. The key point of the type system is that we have the ability to invent such type rules, because we have a mechanism for specifying and statically checking acyclic reachability of regions.

3.2 Static Semantics

We present an overview of the static semantics of ARTS, along with detailed description of some important typing rules. The complete type system can be found in Table 2. In addition to the type system, we define auxiliary functions to lookup and bind classes, types, regions, region definitions, fields and methods in Table 3. We intentionally move all occurrences of substitution into the auxiliary functions to simplify the typing rules, for example, the class lookup for a given type substitutes the actual region arguments into the class definition, and qualifies all of the local region definitions with the type.

We also assume that, for a program to be valid, no identifier can be declared more than once within the same scope. That is, no class name can be declared more than once; no field and method name can be declared more than once within the same class; region names, including formal parameters, cannot be declared more than once within the same class. In fact, we explicitly check this in the case of class names and region names, because we need to check that region definition is well-ordered in the rules.

We use some syntactic shortcuts. By default $E \vdash_\sigma \overline{\sigma}$ represents a possibly empty sequence of judgements $E \vdash_\sigma \sigma_1 \ldots E \vdash_\sigma \sigma_n$. The same abbreviation is used for judgements involving $\overline{fd}$, $\overline{mth}$ and $\overline{t}$. Similarly, $E \vdash_e \overline{e : t}$ stands for $E \vdash_e e_1 : t_1 \ldots E \vdash_e e_n : t_n$, and $[\overline{\sigma/r}]$ abbreviates a sequence of substitutions $[\sigma_1/r_1] \ldots [\sigma_n/r_n]$.

A program P is a pair consisting a fixed sequence of class definitions Π and an expression e. Γ denotes any sequence of class definitions. An environment E may extend a sequence of class definitions with constraints on region names, or the types of variables. The order of the elements in the environment is significant.

$$P ::= \Pi\ e \qquad \Pi ::= \Gamma \qquad \Gamma ::= \emptyset \mid \Gamma, cls \qquad E ::= \Gamma \mid E, x : t \mid E, \overline{\sigma} \rhd r \rhd \overline{\sigma}$$

Because the number of recursively defined regions and types is infinite, to achieve a strict partial order among all possible regions, we identify two requirements that must be satisfied by the type system. Firstly any reachability defined between two regions needs to be one-way only. Secondly, any extension to the reachability relation should not introduce any more reachability between existing regions, but rather introduce new regions together with their reachability relative to existing ones.

Our type system satisfies these two requirements. To ensure one-way reachability, the reachability relation between any two regions can be defined *once* only. This can be achieved by ordering the region definitions so that the later definitions are expressed in terms of earlier ones, as checked by the $\Gamma, \overline{p} \vdash_r \overline{q}$ judgement in the [CLS-DEF] rule.

To satisfy the second requirement, the type system guarantees that if a new region lies between existing regions, then those regions were already related. For example, if $\sigma \rhd \sigma'$ is already defined, then a new region r can be defined via $\sigma \rhd r \rhd \sigma'$, but not $\sigma' \rhd r \rhd \sigma$ because this would introduce cycles into the region structure. Similarly, if there is no existing reachability between σ and σ' can be derived, then $\sigma \rhd r \rhd \sigma'$ is invalid too because it implies $\sigma \rhd \sigma'$ which is unspecified originally. This constraint is enforced by the [REG-DEF] rule which guarantees that the new region variable and the relations on it do not violate the consistency of the region ordering.

Class definition ordering is established by [CLS-DEFS]. Class well-formedness is checked in [CLS-DEF]. Each class defines its own environment E formed from all given class definition, its formal parameter constraints and the type of the current object. Note that we do not put region definitions into E, because we want local regions to be qualified when used in region expressions and types. We recall that the constraints on formal region parameters, $\overline{p}$, are requirements of the class. The well-formedness of the parameter constraints is checked by the judgement $\Pi \vdash_r \overline{p}$; this assumes a whole program context, so that the constraints need not rely just on previously defined regions. Actually we could omit this judgement altogether and rely on checking that the actual region arguments satisfy the constraints whenever the class is used to form a type; including the check on parameter constraints implies that we will reject unusable classes even if they are not used. A region definition q is checked in a context restricted to previously defined classes in Γ, the formal parameter constraints $\overline{p}$ of the current class, and, by the unwinding of [REG-DEFS] earlier local definitions $\overline{q}$, thus guaranteeing the partial ordering of regions, as indicated above. If the class is extended from a supertype then the supertype needs to be valid in the class environment E, both subtype and supertype must live in the same region (their first parameter r_1) to ensure that the region of the current object is not lost through subtyping, and the superclass must be pre-defined in Γ. Finally for a well-formed class, all fields and methods must be well-formed in E.

For a type to be valid by [TYPE], we require the class to be defined in the environment. Its actual region arguments must satisfy the defined constraints by substituting the actual regions into the class definition. If two formal parameters are unconstrained, then they may be bound to any valid region; they may even both be bound to the same region.

Once a well-formed environment is established, it is used to infer region reachability for all valid regions. Region reachability is defined to be transitive; irreflexivity and antisymmetry are consequences of our system. The first three [REACH] rules check the region relations as directly defined by programmers, all other relations are inferred through transitivity rule [REACH-TRANS]. The [REACH-ENV] rule checks defined relations between formal region parameters while the pair of rules [REACH-DEF-FROM/TO] check the defined reachability relations in region definitions.

Another key rule of the type system is the [FIELD] rule where global reachability properties are preserved by placing local restrictions on field references. Fields are static (that is, heap-based) references, so that the field rule needs to ensure that `this` can reference other objects if and only if its region is same to or can reach the regions of the other objects. This is an important invariant of our programs.

In the current setting of our type system, we allow method arguments and result types to freely reference objects in any region. The [METHOD] rule merely checks if the types of arguments and the method body are correct. Dynamic references are considered safe in the sense that they will not form cycles in the object graph as they are local to a method stack. As discussed earlier, stronger constraints could easily be written into this rule to prohibit inter-region dynamic cycles.

3.3 Examples

We give some toy examples to illustrate the use of regions, then show how regions can be used as security levels to capture the ordering of information flow.

Toy Examples

```
class A<p1, p2 from p1, p3>
  a1 from p1 to p2;             // OK
  a2 from p2 to p1;             // BAD require p2 to p1
  a3 from p1 to p3;             // BAD require p1 to p3
  a4 from base;                 // OK

class B<p from base>
  b1 from p;  b2 from b1;  b3 to p;
  A<p, b2, p> f1;               // OK by transitivity
  A<b1, b2, p> f2;              // OK
  A<p, b3, b2> f3;              // BAD require b3 from p
  A<p, base, b1> f4;            // BAD require base from p
  A<p, B<b3>.b2, p> f5;         // BAD require B<b3>.b2 from p

class C<
  p1 from base,                 // OK
  p2 from B<p1>.b1,             // OK p1 already introduced
  p3 from B<p3>.b1,             // BAD p3 undefined yet
  p4 to B<base>.b3,             // BAD require base from base
  p5 from p1 to B<p2>.b2        // OK by transitivity
  >
```

Class `A` shows how regions are defined. The key point in region definitions is that any newly introduced region cannot change the relation of any previous defined region. Class `B` shows that a valid type needs to satisfy its class constraints. Class `C` shows various class constraints on formal region parameters, some with qualified regions.

```
class M<p>
  m to N<p>.n;                  // BAD require ordering of classes

class M'<p>
  m to p;                       // OK

class M"<p1, p2 to N<p1>.n>     // OK do not require ordering of classes
  m to p2;

class N<p>
  n to M<p>.m;
```

The examples above show the importance of class ordering. Class ordering is not only important for correct inheritance, but also important in keeping the ordering of region definitions. In classes `M` and `N`, there is a cycle between `M<p>.m` and `N<p>.n` whenever their class parameters are bound to the same region. To solve this problem, we forbid regions in earlier classes to be defined to relate to regions defined in later classes to enforce the order of region definitions, that is, regions definitions in earlier classes are considered to be earlier than those in later classes. Since class `M` is defined earlier than class `N`, class `M` is not valid and must be rewritten to class `M'` while later class `N` is always valid. Alternatively, class `M` can be rewritten to class `M"` where the region `m` can be defined to reach `N<p1>.n` through the formal parameter `p2` since class ordering does not apply to the constraints on formal region parameters.

```
class SubjectFactory<factory, subject>
  Subject<factory> s1;                // OK
  Subject<subject> s2;                // BAD
  Subject<subject> makeSubject()      // OK
    return new Subject<subject>();    // OK
```

As we discussed earlier our current version of ARTS allows dynamic references (method arguments and results) to be treated differently to static/field references. There is no restriction on what regions can be used for dynamic references. Unrestricted dynamic references add flexibility which can be seen in the factory example. The `SubjectFactory` class models a factory in region `factory` for creating `Subject` objects in region `subject`. Because there is no known reachability between `factory` and `subject`, the type system prevents the `SubjectFactory` class from holding field references to objects in `subject`. However, since dynamic references are not bound by this restriction, new objects can be created in `subject` and returned through the `makeSubject` method for clients to use.

Type Enforced Security Levels. Besides using regions to prevent reference cycles and to reason about aliasing, acyclic regions can also be used as security levels to control access and secure information flow in a multi-level security system (see Section 6). Programmers can express desired access control and information flow policies in the region structure of a program. Because these security policies are now expressed in types, they can be enforced statically by the type system.

```
class Machine<floor>
  display from floor;
  Display<display> disp;
  Machine()
    disp = new Display<display>();
  adjust()                      // modify the display
    ...
class Operator<skill, floor from skill>
  Machine<floor> mach;
  Display<Machine<floor>.display> disp;
  set(Machine<floor> mach)
```

```
    this.mach = mach;
    this.disp = mach.disp;
  operate()                     // do a job, such as adjust the machine
    ...
class Factory<factory>
  skill1 from factory;
  skill2 from factory to skill1;
  floor1 from skill1;
  floor2 from skill2;
  Machine<floor1> mach1;
  Machine<floor2> mach2;
  Operator<skill1, floor1> op1;
  Operator<skill2, floor2> op2;
  Operator<skill2, floor1> op3;
  Operator<skill1, floor2> op4; // BAD, not a valid type
  op1.set(mach1);               // OK
  op1.set(mach2);               // BAD, op1 is on wrong floor
```

A factory has a number of operators and machines. Different machines require different level of skills to operate. In this example, two machines are placed in different floor regions and four operators occupy two skill level regions. The region structure of `Operator` requires an operator to have enough skill to work on a machine. The relations between different skills and floors are defined in the `Factory` class - operators with `skill2` can work on the machines on any floor while operators with `skill1` can only operate the machines on `floor1`. Moreover, operators with `skill1` can never obtain a reference to machines on `floor2`, which implies information stored in region `floor2` can never flow to objects in region `floor1`.

4 Some Properties and a Dynamic Semantics

In this section, first we formalize some static properties about regions for a well-formed program: namely that the region reachability relation is acyclic. Then, after briefly introducing a formal big-step semantics, we characterize the invariants for object references on good heaps for well-formed programs: inter-object references either occur within regions or respect the region reachability relation. Finally we state a standard subject reduction theorem, that, amongst other things, states that heap goodness is preserved through reductions.

First we capture the idea of one region being defined earlier than another. A class definition sequence $\varGamma$, that is well-formed, $\vdash_c \varGamma$, determines a sequence of region definitions. A qualified region $t.r$ that is well-formed, $\varGamma \vdash_\sigma t.r$, is associated with a unique index in the definition sequence, namely that which defines the region name of the qualified region. We call this the *rank* of the region. We also define the rank of `base` to be 0. A region σ with a smaller rank than another σ' is defined earlier; we will also write this as $\sigma \prec \sigma'$. If a region appears as a bound in a definition for another, then the regions must have different ranks; essentially this is because of the [REG-DEF] rule.

Table 4. Dynamic Features

$\iota, \iota_t \in$ TypedLocation		
$e \in$ Expression	$::=$	$... \mid \iota$
$v \in$ Value	$::=$	$\iota \mid$ `null`
$obj \in$ Object	$=$	FieldName $\longrightarrow$ Value
$H \in$ Heap	$=$	TypedLocation $\longrightarrow$ Object

We are now in a position to state a fundamental property of reachability proofs: any reachability between two regions can be defined through a sequence of region definitions, where the successive ranks are strictly decreasing until a minimum rank is reached, after which the ranks are strictly increasing.

Lemma 1 (Reachability via Earlier Regions). *Given $\vdash_c \Gamma$ and $\Gamma \vdash_\sigma \sigma, \sigma'$: If $\Gamma \vdash_\triangleright \sigma \triangleright \sigma'$ then $\exists \sigma_1 ... \sigma_n$ for $n > 1$ such that:*

1. *$\sigma \equiv \sigma_1 \triangleright ... \triangleright \sigma_n \equiv \sigma'$, and*
2. *$\sigma_1 \succ \sigma_2 \succ ... \succ \sigma_m \prec ... \prec \sigma_n$ for some $m \in 1..n$ where $(\sigma_i \triangleright \sigma_{i+1}) \in \mathcal{Q}(t_i)$ for $1 \leq i < m$ (where $\sigma_i \equiv t_i.r$), and $(\sigma_i \triangleright \sigma_{i+1}) \in \mathcal{Q}(t_{i+1})$ for $m \leq i < n$.*

Proof Outline. Any proof of reachability must construct a sequence of one or more applications of a definition via [REACH-DEF-FROM/TO]. So the first part of the lemma follows, with successive pairs belonging to some definition. Suppose the second part does not hold. Then we can find $(\sigma_{i-1} \triangleright \sigma_i \triangleright \sigma_{i+1}) \in \mathcal{Q}(t_i)$ with $(\sigma_{i-1} \prec \sigma_i \succ \sigma_{i+1})$. But by the requirement of [REG-DEF] that any reachability of constraints can be derivable from earlier definitions, we see that we can omit σ_i from our proof. The result follows by an induction on the maximum region definition rank in Γ.

Theorem 1 (Acyclicity of Regions). *Given $\vdash_c \Gamma$ and $\Gamma \vdash_\sigma \sigma, \sigma'$: if $\Gamma \vdash_\triangleright \sigma \triangleright \sigma'$ then $\Gamma \nvdash_\triangleright \sigma' \trianglerighteq \sigma$.*

Proof Outline. Suppose, by contradiction that $\Gamma \vdash_\triangleright \sigma \triangleright \sigma'$ and $\Gamma \vdash_\triangleright \sigma' \trianglerighteq \sigma$. Without loss of generality, assume that $\sigma \prec \sigma'$. By transitivity of $\triangleright$, we find that $\Gamma \vdash_\triangleright \sigma \triangleright \sigma$. From Lemma 1 we can see that we can find two (possibly same) earlier regions σ_1 and σ_2, such that $\sigma \triangleright \sigma_1 ... \sigma_2 \triangleright \sigma$ and $\sigma_1, \sigma_2 \prec \sigma$. But again, we must have $(\sigma_2 \triangleright \sigma \triangleright \sigma_1) \in \mathcal{Q}(t)$, so it follows that $\sigma_2 \triangleright \sigma_1$. The result follows by an induction on the minimum rank of the two regions under consideration.

Let us now consider the dynamic semantics. Table 4 formulates some dynamic features of ARTS and the dynamic semantics is given in Table 5. Table 6 shows the rules for well-formedness of heap and expression in the dynamic model.

We incorporate full type information (with regions) with the locations of the heap, rather than in the objects. These help to simplify the semantics and the proof of dynamic properties. Note that none of the reduction behavior depends on this type information; the `new` t reduction only uses the field names of the class; method dispatch only depends on the class and not on the region bindings

Table 5. Dynamic Semantics

$$\frac{\iota_t \notin dom(H) \quad \mathcal{F}(P,t) = \overline{\_\ f} \qquad H' \equiv H, \iota_t \mapsto \overline{f \mapsto \texttt{null}}}{H, \texttt{new}\ t \Downarrow \iota_t, H'}\ \text{[RED-NEW]} \qquad \frac{H, e \Downarrow \iota, H'}{H, e.f \Downarrow H(\iota)(f), H'}\ \text{[RED-FIELD]}$$

$$\frac{H, e \Downarrow \iota, H' \quad H', e' \Downarrow v, H'' \quad H''' \equiv H''[\iota \mapsto H''(\iota)[f \mapsto v]]}{H, e.f = e' \Downarrow v, H'''}\ \text{[RED-ASSIGN]} \qquad \frac{H, e \Downarrow \iota_t, H' \quad H', \overline{e} \Downarrow \overline{v}, H'' \quad \mathcal{M}(P,t,m) = (\_, \overline{x}, \_, e') \quad H'', e'[\iota_t/\texttt{this}, \overline{v/x}] \Downarrow v, H'''}{H, e.m(\overline{e}) \Downarrow v, H'''}\ \text{[RED-CALL]}$$

$$\frac{H, e \Downarrow \_, H' \quad H', e' \Downarrow v, H''}{H, e; e' \Downarrow v, H''}\ \text{[RED-SEQ]} \qquad \frac{H, e \Downarrow \iota, H' \quad H', e' \Downarrow v, H''}{H, \texttt{if}\ e\ e'\ e'' \Downarrow v, H''}\ \text{[RED-IF-LOCATION]} \qquad \frac{H, e \Downarrow \texttt{null}, H' \quad H', e'' \Downarrow v, H''}{H, \texttt{if}\ e\ e'\ e'' \Downarrow v, H''}\ \text{[RED-IF-NULL]}$$

Table 6. Auxiliary Rules for Dynamic Semantics

$$\frac{\forall \iota_t \in dom(H) \cdot \quad \Gamma \vdash_t t \quad H(\iota_t) = \overline{f \mapsto v} \quad \mathcal{F}(t) = \overline{t\ f} \quad \Gamma \vdash_e \overline{v : t}}{\Gamma \vdash_H H}\ \text{[HEAP-WELLFORMED]} \qquad \frac{\Gamma \vdash_t t}{\Gamma \vdash_e \iota_t : t}\ \text{[EXPR-LOCATION]}$$

of the target object. Again, for simplicity, we do not use local variables in our model, so we use substitution of method arguments into method bodies, thus avoiding the extra machinery of a stack frame. Note that the `if` test branches on `null` test value.

Now a well-formed heap ensures that any field of an object in the heap stores the value whose actual type respects the declared type of the field in the type of the object. This leads directly to the following property for good heaps, whose proof follows directly from the static properties of regions. This states that object references respect region reachability. Consequently, reference cycles must occur within regions.

Lemma 2 (Direct Referenceability). *Given $\vdash_c \Gamma$ and $\Gamma \vdash_H H$: if $\iota_{c\langle\sigma...\rangle} \mapsto [...\ \_ \mapsto \iota'_{c'\langle\sigma'...\rangle}\ ...] \in H$, then $\Gamma \vdash_\triangleright \sigma \trianglerighteq \sigma'$.*

Proof. By the [HEAP-WELLFORMED] rule, $c\langle\sigma...\rangle$ must be well-formed and $c'\langle\sigma'...\rangle$ is the type of one of class c's fields. By the [FIELD] rule, $\Gamma \vdash_\triangleright \sigma \trianglerighteq \sigma'$.

Theorem 2 (Reachability and Cycles). *Given $\vdash_c \Gamma$ and $\Gamma \vdash_H H$: if $\iota_{c\langle\sigma...\rangle}$ can reach $\iota'_{c'\langle\sigma'...\rangle}$ through a path of direct references in the heap H, then $\Gamma \vdash_\triangleright \sigma \trianglerighteq \sigma'$. Furthermore, if there is a path in H in the reverse direction, then $\sigma = \sigma'$.*

Proof. By Lemma 2 for each direct reference in the path, and the transitivity of $\triangleright$, the first part follows. When there is a reverse path, the second part follows by the acyclicity of $\triangleright$, as in Theorem 1.

Finally we present a standard subject reduction result, together with a statement that goodness of a heap is invariant through expression reductions. This implies that the heap invariants are maintained through program execution.

Theorem 3 (Preservation). *Given* $\vdash_c \Gamma$, *and* $\Gamma \vdash_H H$:

$$\textit{if} \begin{cases} \Gamma \vdash_e e : t \\ H, e \Downarrow v, H' \end{cases} \quad \textit{then} \begin{cases} \Gamma \vdash_e v : t \\ \Gamma \vdash_H H'. \end{cases}$$

Proof Outline. The proof for type preservation is completely standard by structural induction on the form of expressions over reduction rules. We do not show a subtype of t for v, because it is covered by subsumption.

5 Discussion

5.1 Expressiveness and Limitation

ARTS provides a powerful framework for allowing a succinct description of many classes of reachability relations. It employs an intuitive notion of region to capture the concept of strongly connected components in graph theory, which appears to be natural and flexible enough to express various data structures in programs. The type checking is simple yet efficient and powerful; it locally checks programmer-defined reachability relations to guarantee global acyclicity so that separate compilation can be allowed. ARTS allows the programmer to name any possible region even though the number of regions may be unbounded. The ability to multiply instantiate region definitions through type qualification gives programmers enough choice to identify as many distinct regions as they need.

ARTS can significantly improve program understanding. Programmers are able to specify via types whether cycles are allowed or disallowed. ARTS can also be used to reason about aliasing because regions are disjoint and objects live in a single region for their lifetime. Moreover, ARTS can express some information flow policy (see Section 6) and even encapsulate objects (see later this Section). ARTS has direct application in multi-threaded programs. Multi-threading will not affect the structure of the object graph, but knowledge of the region structure allows, for example, ordered locking strategies to be imposed [4]. However, in this paper we only consider the fundamental issues in reachability and acyclicity in the object graph, and do not cover the issues with multi-threading.

Of course, as with any type system, there is a price to pay for the improved safety offered by strong type checking. First, there is the extra syntactic weight associated with more expressiveness; the syntax burden is not too taxing, amounting to the cost of parameterized types. For our purposes, it is essential to distinguish between type (schema) definition and the use of a type (instance). Without this distinction, our proposal would provide little more than the name-based access restrictions offered by module or package-based approaches. Second and more importantly, what are the expressive limitations of our approach? In some sense, none, because the type system proposed in this paper allows programmers to code with no structural constraint whatsoever, that is, all objects live in the same region. In Section 5.2 we will discuss the ability to integrate with region-free code.

Realistically though, in order to benefit from the ability of our type system to inhibit cycles and/or sharing, it is necessary to make inhibiting design decisions. Our type system will insist that programmers decide which object fields may form part of a cycle, and which may not. It is relatively simple then to record types which will cause the design decision to be enforced. Again as with any type system, there is a trade-off between extra safety offered by strong type checking, and the loss of flexibility in the programming model, or at least annoyance at being made to impose restrictions early on in a design. In practice our system will not be too annoying, because when programmers do not care about cycles, they can effectively allow them to occur anywhere, and the appropriate types are the least complex to express, corresponding exactly to the marked up legacy code.

ARTS can express recursive data structures. However, such a structure needs to have fields with the same type as the self type. Because they have the same type, all the structural objects forming the recursive data structure must all live in the same region. As a result, all the data objects will also live in the same region as each other. ARTS allows the programmer to name any possible region even though the number of regions is unbounded. This flexibility complicates the task of type checking on the reachability relations. In the worst case, to check the relation between two regions, the type checker may have to look into all classes in the recursive type qualification steps for both regions and for each class the type checker may have to look into all region definitions of the class. However, in practice both the number of recursive steps to make sensible use of a region and the number of region definitions in a class are very small, so that the runtime for type checking should not be significant. Moreover, any reachability relation is decidable because each region can only have a finite number of recursive steps and each class can only have a finite number of region definitions.

5.2 Extensions to the Core Language

Object Encapsulation with Owned Regions. Our type system can provide region existential polymorphism at almost no cost. Existential regions can provide the same level of object encapsulation as ownership types do (see Section 6). In our extended language, regions can be hidden by using an *owned* declaration; owned regions cannot be named via a type qualification and hence local to the scope of the class. To enable type checking on owned regions, we simply need to add an optional key word `owned` in front of the region name in the syntax when a region is introduced, and disable type qualification over owned regions in the region rules.

It is important that owned regions are instance level and are encapsulated by their defining objects because no one can name them from outside. This is similar to the internal context `this` in ownership types. An owned region can only be named within the defining object or propagated to its encapsulation via region parameters. Unowned regions remain static to types, but may not be named globally if their types are parameterized by an owned region. This results in ownership-like structures on some parts of the DAG structure of our object model. Programmers may have better understanding and fine-grained control over the reference structure.

For example, owned regions can be used to express ownership-like linked lists without suffering the long-recognized problem of expressing iterators in ownership types. In the next example, every list object now has its own implementation encapsulated by the owned region `link`. Because owned regions are instance-based, they are not shared by different list objects of the same type (like unowned regions are). The link objects are owned by the list because no object from outside of the list can name the owned region `link`. An iterator can access the internal data of the list, because it is created inside the list, but can still be used from outside because it lives in the same region as the list object. We use subtyping to hide the name of the `link` region in the type of the iterator, so the client can give a type for the iterator without knowing the right name for the `link` region.

```
class List<list, data from list>
  owned link from list to data;
  Link<link, data> head;
  Link<link, data> tail;
  Iterator<list, data> getIterator()
    return new ListIterator<list, data, link>(head);
  ...
class Iterator<list, data from list>
  ...
class ListIterator<list, data from list, link from list to data>
      extends Iterator<list, data>
  Link<link, data> current;
  Iterator(Link<link, data> current)
    this.current = current;
  ...
```

Default Regions and Interoperability with Legacy Code. The burden of region annotations and the interoperability with legacy code may affect the ease of use for the language. In practice, regions are only used when needed. We expect programmers to use regions in some critical sections, for example, to protect crucial class invariants from unexpected method reentrant calls. In many cases regions can be circumvented when they are not needed and programmers need not even be aware of regions. We design a few region defaults to help reduce the number of region annotations and integrate non-region code such as legacy library classes. The compiler may automatically annotate regions with the default policy.

For classes that do not have formal region parameters, the key word `here` is used to refer the region the current object lives in. This is similar to `this`, `self` or me used in many object-oriented languages to refer the current object. For bindings of regions in types, if a type is declared without any region, i.e. only a class name, then a default region is bound to all formal parameters of the class. In the main routine, base can be used as the default region for types while in classes the first formal parameter of the class (or here if there is none) is the default region. For legacy code all formal region parameters and type declarations are defaulted.

Flexible Class Constraints. Our language can be easily extended with more possible relations other than acyclic reachability. Of course, these possible relations can only be used to constrain formal region parameters of a class, i.e., not to the region definitions. They are just class constraints for valid region parameter bindings, they should not be confused with the reachability relation defined between actual regions. The simplest is to allow reachability from one region parameter to another to be declared without forbidding backwards reachability, denoted as $\trianglerighteq$ in the type system which is a reflexive closure of the acyclic reachability $\triangleright$. When declared, it means that we can allow references in the direction of the declared reachability, but we can allow both parameters to be bound to the same actual region. This is a somewhat trivial but useful extension for more flexible programming.

6 Related Work

To our knowledge, this is the first attempt to reason about cycles and sharing in programs based on a type system imposing reachability constraints on the object graph. Our type system does have some similarities with other type systems, such as parametric polymorphism and existential polymorphism on regions. Our type qualified regions and region parameterized types provide a novel way to support global naming and static reasoning on an unbounded number of regions.

Ownership Type Systems. Many type systems focus on alias management and attempt to restrict references into a limited scope. Early work like Islands [16] and Balloons [2] enforced full encapsulation on objects which prevented referencing across the encapsulation. They are generally too restrictive. Universes [19] improved the expressiveness of full encapsulation by introducing read-only references to cross the boundary of encapsulation. Ownership types [7, 6, 5] improved the previous work on object level encapsulation by allowing unrestricted outgoing references from an encapsulation while still preventing incoming referencing into an encapsulation.

Ownership types use parameterized type systems to pass the names of objects via class parameters. In order to declare a type for a reference, one must be able to name the 'owner' that encapsulates this object. Encapsulation is protected from incoming referencing because the owners of objects inside an encapsulation cannot be named from the outside. However, objects inside an encapsulation are able to name the objects living outside through the owner names passed in as class parameters. Our type system is close to ownership types that are parameterized classes in a similar way, and the first parameter identifies the owner/region of the `this` object. However, the invariant of our system is about acyclicity rather than encapsulation. The major difference between these two properties is that encapsulation is instance-based and enforced through ownership, which is a local property of an object whereas acyclicity is global.

A variant of ownership types has recently been proposed [1] to allow programmers to specify aliasing policy between ownership domains (which partition an owner's context) rather than ownership types' owner-as-dominator property. Their domains and link polices are defined within classes in a similar fashion to our regions and region constraints. Their aliasing policies define referenceability, the ability to directly reference an object, between domains. They are concerned with neither reachability – the link policies are not transitive, nor acyclicity – it is possible to link domains cyclically. In fact, our regions can also be used to reason about aliasing in a similar way but with a different policy – we can enforce a deep reachability policy instead of a shallow direct referenceability policy.

SCJ [4] introduced a concept of lock level to help order locks statically, extending the idea of using ownership types to identify those objects which are not shared by threads (so require no locks). In their language, lock levels are partially ordered and all locks are partitioned into lock levels and therefore ordered according to their lock levels. Similar to ownership domains and our regions, their lock levels and ordering are defined within classes. However their lock levels are static to classes which means the number of all lock levels is limited by the number of classes. Moreover, their published type system does not appear to check the partial ordering of lock levels.

In contrast, our system supports region parametric polymorphism for code reuse and region existential polymorphism for object level encapsulation. We also provide a novel type-based naming mechanism to allow an unbounded number of regions to be named throughout the system. Type qualified regions allow completely static reasoning on any possible region and their constraints, excluding owned regions which only can be reasoned about locally. This gives a richer model for our region structure, and means that a programmer is able to be more discriminating in choosing an appropriate region structure. We have proved that our type system guarantees acyclicity amongst an unbounded number of regions.

Pointer and Shape Analysis. Pointer analysis attempts to acquire knowledge about run-time pointers via whole program analysis and uses this information to help program understanding and optimization [15]. Shape analysis is built on the top of pointer analysis to identify the shapes of data structures [10, 26]. Hackett and Rugina have recently proposed a shape analysis algorithm that breaks down global analysis about entire heap into local analysis about smaller memory abstractions which they also call regions [12]. Similar to our regions, theirs are disjoint sets of memory locations, which allow the subsequent shape analysis to safely conclude that an update in one region will not change the values of locations in other regions. Different from ours, their memory regions are not acyclic and they simply identify direct *points-to* relations between regions rather than transitive reachability relations.

Compared to type systems, pointer and shape analysis require little or no language annotation. Proponents of this approach often consider type systems are too restrictive and may rule out some good programs unnecessarily. However, exact pointer and shape analysis not only require exponential time for verification, but are undecidable, so in practice, may be of relatively low precision. They are also hard to scale to large or incomplete programs. Moreover, most work in this area has been done for C-like languages, and less for object-oriented languages.

In order to improve accuracy of program analysis, 'define for analyzability' approaches, such as ADDS and ASAP [14, 17], ask the programmer to explicitly describe some properties of data structures. They use the concept of dimension which is related to the depth in a linked list or a tree. ARTS's region concept is different, but the knowledge of regions may also be useful to allow more accurate program analysis and better performance, especially for cyclic data structures.

Type-Based Information Flow Security. ARTS is primarily designed to reason about reference reachability and confine reference cycles. But it turns out to have direct application in the area of information flow security. This is because the partially ordered acyclic regions are conceptually same to the classic lattice model of information flow control by Denning and Denning [8, 9].

Type systems have already been used to secure information flow within programs in a multi-level security system based on the lattice model. The general idea is that every type is associated with a particular security class [9, 24, 25]. Security classes, sometimes also called security groups, roles, users or principals, form a lattice which is a presumed finite set partially ordered by the level of security (high or low). Information flow operations, such as assignments, must respect the order of the security classes attached to types.

Some type systems, instead of associating types with a single security class, they allow multiple security classes to form a security property or label for each type [13, 20]. These access control list like security labels allow more complex security control and dynamic manipulation. The labels are pre-ordered rather than partially ordered. The correctness of information flow still relies on the order of predefined security classes.

ARTS is similar to these type systems in the way types are formed with regions corresponding security classes. However, the soundness of these systems depends on the predefined finite security classes which are assumed to be partially ordered, i.e., they do not actually check the correctness of security classes and their ordering. The type checking only ensures that programs respect the security policies embedded in the types. Besides the similar treatment of types, ARTS allows the programmer to specify desired security classes and the security level between them. These programmer-defined security classes are infinite and guaranteed to be partially ordered via type checking. This is generally a harder task for modular type checkers because global acyclic invariant is ensured by checking local reachability relations. We prove our type system is sound.

Most work on information flow security has been done in procedural languages, which mainly deal with primitive data types. They allow data in a low security class to be assigned to a variable in a high security class, but not vice versa. In the simple object-oriented language we present in this paper, we only have object types, and security classes are bound to class parameters. Because objects have fields, the security class has be to invariant for assignments. Otherwise, we could employ some simple covariant mechanisms such as declared covariant on class parameters like JFlow does [20] or more powerful mechanisms such as variant parametric types [18].

Region-Based Memory Management. Our notion of region is similar to that used in region-based memory management [22, 23, 11] because they both refer to a partition of data objects. Region-based memory management focuses on the safety and efficiency of explicit memory allocation and deallocation on the basis of regions. The ordering relation on regions is based on lifetimes, that is, on which regions may outlive others. Instead our regions represent a static abstraction of strongly connected components in object graphs and focus on reference cycles. Although the region structures are different, it would be interesting to see if objects that live in our regions with cyclic references share the same lifetime. Moreover the outlive relationship between regions of memory needs to be acyclic as well, where our concept of regions may just fit in.

7 Conclusion and Future Work

The major result of this paper is a class-based region-parametric type system that allows programmers to specify regions which trap all object reference cycles, and to otherwise control the acyclic reachability for all objects. This provides a novel contribution to ongoing work investigating the use of type systems, and other formalisms, for taming arbitrary object reference structures. There are fruitful avenues opened up for ongoing research. For us the most promising direction is to investigate incorporating more kinds of constraints, such as possible sharing, non-sharing, and ownership-like containment properties. It is still unclear to us whether attempting to combine a number of such kinds of constraints will be intractable, both in terms of the syntactic load, and the semantic complexity brought about by the interactions between various kinds of constraints. We remain hopeful that by pursuing these ideas from a graph theoretic viewpoint, more fruitful and expressive approaches will surface.

References

1. J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *European Conference for Object-Oriented Programming (ECOOP)*, July 2004.
2. P. S. Almeida. Balloon types: Controlling sharing of state in data types. *Lecture Notes in Computer Science*, 1241:32–59, 1997.

3. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. In S. Eisenbach, G. T. Leavens, P. Müller, A. Poetzsch-Heffter, and E. Poll, editors, *Formal Techniques for Java-like Programs (FTfJP)*, July 2003. Published as Technical Report 408 from ETH Zurich.
4. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.
5. D. Clarke. *Object Ownership and Containment*. PhD thesis, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia, 2001.
6. D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In *European Conference for Object-Oriented Programming (ECOOP)*, 2001.
7. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 48–64. ACM Press, 1998.
8. D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.
9. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.
10. R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–15. ACM Press, 1996.
11. D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 282–293. ACM Press, 2002.
12. B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL*, pages 310–323, 2005.
13. N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In ACM, editor, *Conference record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California, 19–21 January 1998*, pages 365–377, New York, NY, USA, 1998. ACM Press.
14. L. J. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, volume 27, pages 249–260, New York, NY, 1992. ACM Press.
15. M. Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61. ACM Press, 2001.
16. J. Hogg. Islands: aliasing protection in object-oriented languages. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 271–285. ACM Press, 1991.
17. J. Hummel, L. J. Hendren, and A. Nicolau. A language for conveying the aliasing properties of dynamic, pointer-based data structures. In *8th International Parallel Processing Symposium*, pages 208–216, Cancun, Mexico, 1994.

18. A. Igarashi and M. Viroli. On variance-based subtyping for parametric types. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 441–469. Springer-Verlag, 2002.
19. P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. *Programming Languages and Fundamentals of Programming*, 1999.
20. A. C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.
21. K. Rustan, M. Leino, and P. Müller. Object invariants in dynamic contexts. In *European Conference for Object-Oriented Programming (ECOOP)*, 2004.
22. M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Symposium on Principles of Programming Languages*, pages 188–201, 1994.
23. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 1997.
24. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.
25. D. M. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT*, pages 607–621, 1997.
26. R. Wilhelm, S. Sagiv, and T. W. Reps. Shape analysis. In *Computational Complexity*, pages 1–17, 2000.

Eclat: Automatic Generation and Classification of Test Inputs

Carlos Pacheco and Michael D. Ernst

MIT Computer Science and Artificial Intelligence Lab,
The Stata Center, 32 Vassar Street,
Cambridge, MA 02139 USA
{cpacheco, mernst}@csail.mit.edu

Abstract. This paper describes a technique that selects, from a large set of test inputs, a small subset likely to reveal faults in the software under test. The technique takes a program or software component, plus a set of correct executions—say, from observations of the software running properly, or from an existing test suite that a user wishes to enhance. The technique first infers an operational model of the software's operation. Then, inputs whose operational pattern of execution differs from the model in specific ways are suggestive of faults. These inputs are further reduced by selecting only one input per operational pattern. The result is a small portion of the original inputs, deemed by the technique as most likely to reveal faults. Thus, the technique can also be seen as an error-detection technique.

The paper describes two additional techniques that complement test input selection. One is a technique for automatically producing an oracle (a set of assertions) for a test input from the operational model, thus transforming the test input into a test case. The other is a classification-guided test input generation technique that also makes use of operational models and patterns. When generating inputs, it filters out code sequences that are unlikely to contribute to legal inputs, improving the efficiency of its search for fault-revealing inputs.

We have implemented these techniques in the Eclat tool, which generates unit tests for Java classes. Eclat's input is a set of classes to test and an example program execution—say, a passing test suite. Eclat's output is a set of JUnit test cases, each containing a potentially fault-revealing input and a set of assertions at least one of which fails. In our experiments, Eclat successfully generated inputs that exposed fault-revealing behavior; we have used Eclat to reveal real errors in programs. The inputs it selects as fault-revealing are an order of magnitude as likely to reveal a fault as all generated inputs.

1 Introduction

Much of the skill in testing a software artifact lies in carefully constructing a small set of test cases that reveals as many errors as possible. A test case has two components: an *input* to the program or module, and an *oracle*, a procedure that determines whether the program behaves as expected on the input. Many techniques can automatically generate candidate inputs for a program [10, 18, 17, 23, 8, 4, 19, 9, 12], but constructing an oracle for each input remains a largely manual task (unless a formal specification of the software exists, which is rare). Thus, a test engineer wishing to use automated input generation techniques is often faced with the task of inspecting each resulting candidate input, determining whether it is a useful addition to the test suite, and writing an oracle for the input or somehow verifying that the output is correct. Doing so for even a few dozen inputs—much less the thousands of inputs automated techniques can generate—can be very costly in manual effort.

This paper presents three techniques that help the tester with the difficult task of creating new test cases. The first technique is an input selection technique: it selects, from a large set of test inputs, a small subset likely to reveal faults in the software under test—inputs for which writing full-fledged test cases is worth the effort. The goal of the technique is to focus the tester's effort on inputs most likely to reveal faults. Thus, the technique can also be viewed as an error-detection technique, and we have used it to find real errors in practice.

The input selection technique works by comparing the program's behavior on a given input against an operational model of correct operation. The model is derived from an example program execution, which can be an initial test suite or a set of program runs. If the program violates the model when run on the input, the technique classifies the input as (1) likely to constitute an illegal input that the program is not required to handle, (2) likely to produce normal operation of the program (despite violating the model), or (3) likely to reveal a fault. A second component of the technique (called the reducer) discards redundant inputs—inputs that lead to similar program behavior.

The other two techniques complement the input selection technique, by converting its output (test inputs) into a test suite (consisting of full-fledged test cases), and by providing a source of candidate test inputs for it to operate on.

Converting a test input into a test case requires the addition of an oracle, which determines whether the test succeeds or fails. We use an oracle that checks the properties in the operational model. Since the model was derived from correct executions, those properties are suggestive of correct behavior. By construction, the selected inputs will fail on these oracles. Together, the input selection and oracle generation techniques produce a set of failing test cases. This is a great starting point for the tester, whose job is to inspect each input, determine if its execution is in fact faulty, and determine if the oracle captures the proper behavior of the input. The tester can accept, reject, or modify each test input and test oracle.

The third technique is a generation-guided test input generation technique that makes use of operational-model-based classification to construct legal inputs. The input selection technique requires a set of candidate inputs; this technique provides it, while avoiding the generation of many illegal inputs.

We have implemented these techniques in the Eclat tool, which generates unit tests for Java classes. Eclat's input is a set of classes to test and an example program execution (say, a passing test suite). Eclat's output is a set of JUnit test cases, each containing a potentially fault-revealing input and a set of assertions at least one of which fails. Our experiments show that Eclat reveals real errors in programs, and the inputs it selects are an order of magnitude as likely to reveal a fault as all generated inputs. Eclat is publicly available at `http://pag.csail.mit.edu/eclat/`.

The rest of the paper is structured as follows. Section 2 introduces the techniques with an example use of Eclat, a tool that implements them. Section 3 describes the techniques in detail. Section 4 describes the Eclat tool. Section 5 details our experimental evaluation of the technique. Section 6 discusses related and future work, and Section 7 concludes.

2 Example: BoundedStack

We illustrate the test generation and selection technique by describing the operation of the Eclat tool, when applied to a bounded stack implementation used previously in the literature [22, 30, 9]. The bounded stack implementation (Figure 1) and testing code were written in Java by two students, an "author" and a "tester." The tester wrote a set of axioms on which the author based the implementation. The tester also wrote two small test suites by hand (one containing 8 tests, the other 12) using different methodologies [22]. The smaller test suite reveals no errors, and the larger suite reveals one error (the method `pop` incorrectly handles popping an empty stack).

Eclat takes two inputs: the class under test, and a set of correct uses, in the form of an executable program that exercises the class. In this example, the set of correct uses is the 8-test passing test suite.

```
public class BoundedStack {
  private int[] elems;
  private int numElems;
  private int max;

  public BoundedStack() { ... }
  public int getNumberOfElements() { ... }
  public int[] getArray() { ... }
  public int maxSize() { ... }
  public boolean isFull() { ... }
  public boolean isEmpty() { ... }
  public boolean isMember(int k) { ... }
  public void push(int k) { ... }
  public int top() { ... }

  public void pop() {
    numElems --;
  }

  public boolean equals(BoundedStack s) {
    if (s.maxSize() != max)
      return false;
    if (s.getNumberOfElements() != numElems)
      return false;
    int[] sElems = s.getArray();
    for (int j=0; j<numElems; j++)  {
      if (elems[j] != sElems[j])
        return false;
    }
    return true;
  }
}
```

Fig. 1. Class `BoundedStack` [22] (abbreviated). Methods `pop` and `equals` contain errors

Eclat Report

Input 1
```
BoundedStack var8 = new BoundedStack();
var8.push(2);
int var9 = var8.getNumberOfElements();
var8.push(var9);
```
The last method invocation violated this property:

On exit: $size(\texttt{var8.elems}[]) - 1 \neq \texttt{var8.elems}[\texttt{var8.max} - 1]$

During execution of the last method invocation, a postcondition was violated. Since no preconditions were violated, this suggests a fault.

Input 2
```
BoundedStack var8 = new BoundedStack();
var8.equals((BoundedStack)null);
```
The last method invocation signaled a `java.lang.NullPointerException`.

There were no violations, but a throwable was signaled. Since the throwable is considered severe, this suggests a fault.

Input 3
```
BoundedStack var8 = new BoundedStack();
var8.pop();
```
The last method invocation violated this property:

On exit: $\texttt{numElems} \geq 0$

During execution of the last method invocation, a postcondition was violated. Since no preconditions were violated, this suggests a fault.

Fig. 2. Eclat's XML output for `BoundedStack` (formatted for presentation). Inputs 2 and 3 expose errors in the code under test. Input 1 is a false report: it merely indicates a deficiency in the original test suite

Eclat's output is a set of 3 new inputs—uses of the stack—that are classified as fault-revealing by the tool because their behavior differs from the provided test suite. Eclat can produce output in text, XML, or a JUnit test suite. Figure 2 shows the output in XML form. Each input is accompanied by an explanation of why the input suggests a fault, including any violated properties. Each violated property was true during execution of the original test suite, but was violated by the new input.

Input 1 violates one property during the call of `var8.push(var9)`. The violated property says that the last element of array `elems` is never equal to its index. This input reveals no fault; Eclat has made a mistake. The input, however, does point out a stack state not covered by the original test suite, so it may be a good addition to the test suite.

Execution of Input 2 violates no properties, but the `equals` method throws an exception. Eclat classifies the input as fault-revealing. The `equals` method (Figure 1) incorrectly handles a `null` argument. This fault went undetected in all previous analyses of the class [22, 30, 9].

```
public void test_3_pop() throws Exception {

  ubs.BoundedStack var8 = new ubs.BoundedStack();

  // Check preconditions.
  checkPreconditions_pop(var8);
  checkObjectInvariants(var8);

  var8.pop();

  // Check postconditions.
  checkPostconditions_pop(var8);
  checkObjectInvariants(var8);

}

public static void checkPreconditions_pop(Object thiz) {

  // Check: elems[max-1] >= 0
  junit.framework.Assert.assertTrue(
    eclat.Helper.intArray(this, "elems")[eclat.Helper.intField(this,"max")-1] >= 0);
}

public static void checkPostconditions_pop(Object thiz) {

  // Check: elems[max-1] >= 0
  junit.framework.Assert.assertTrue(
    eclat.Helper.intArray(this, "elems")[eclat.Helper.intField(this,"max")-1] >= 0);
}

public static void checkObjectInvariants(Object thiz) {

  // Check: max == elems.length
  junit.framework.Assert.assertTrue(
    eclat.Helper.intField(thiz, "max")
    == eclat.Helper.intArray(thiz, "elems").length);

  // Check: elems != null
  junit.framework.Assert.assertTrue(
    eclat.Helper.intArray(thiz, "elems") != null);

  // Check: max == 2
  junit.framework.Assert.assertTrue(
    eclat.Helper.intField(thiz, "max") == 2);

  // Check: numElems >= 0
  junit.framework.Assert.assertTrue(
    eclat.Helper.intField(thiz, "numElems") >= 0);
}
```

Fig. 3. JUnit test created by Eclat corresponding to Input 3 of Figure 2. When this JUnit test is executed, the last assertion in `checkObjectInvariants` fails during the second call (at the end of `test_3_pop`). This test detects an error in `BoundedStack`'s handling of `pop` when applied to an empty stack. Fields like `this.elems` are accessed via reflection, through method calls like `eclat.Helper.intArray(this, "elems")`. This allows the JUnit test suite to access non-public members of the tested class

Input 3 is classified as fault-revealing because its execution violates the property `numElems` $\geq$ 0. The variable `numElems` becomes negative after a call of `pop` on an empty stack. Eclat has revealed another true error: the `pop` method always decrements the top-of-stack pointer, even on an empty stack. This is a subtle error, because it silently corrupts the stack's state, and a fault only arises on a subsequent access to the stack. In particular, Input 3 itself has no user-observable fault; Eclat detects the corrupted stack state before it leads to an observable fault. A more complicated input—for example, an input that attempts to push an element when the stack pointer is negative and leads to an out-of-bounds exception—would probably be harder to understand and less useful for debugging.

Figure 3 shows a portion of Eclat's JUnit output. The figure shows the JUnit test created for Input 3, and its associated helper methods. Each test in the JUnit test suite will fail upon execution, indicating the violated property.

In summary, Eclat creates 3 inputs that quickly lead a user to discover two errors, and provides a JUnit test suite that exhibits the faulty behavior. Behind the curtains, Eclat generates and analyzes 806 distinct inputs. Some are discarded because they violate no properties and throw no exceptions (and thus suggest no faults). Some are discarded because they violate properties but are determined to constitute illegal uses of the class instead of faults. Some are discarded because they violate properties but are considered a new but non-faulty use of the class. Finally, some inputs are discarded because they behave similarly to already-chosen inputs: 5 of the inputs expose the pop-on-empty-stack fault (for example, one input pushes two items and then pops three times) but only one is selected.

3 Selection and Generation via Classification

This section describes the technique for selecting test inputs likely to reveal faults (Sections 3.1–3.3), the use of an operational model to create test cases from test inputs (Section 3.4), and the technique for generating candidate inputs (Section 3.5). We describe the techniques in the context of unit testing in an object-oriented programming language. The techniques can also be applied to non-object-oriented programs and to components larger than methods and constructors (see Section 3.6).

Figure 4 shows the input selection technique. The technique requires three things: (1) the program under test, (2) a set of correct executions of the program (for instance, an existing passing test suite for the program that a user wishes to enhance), and (3) a source of candidate inputs (each candidate may be an illegal input, or cause the program to behave normally, or reveal a fault).

The selection technique has three steps.

- **Model generation.** Observe the program's behavior on the provided correct executions, and create an *operational model* of correct behavior (Section 3.1).
- **Classification.** Classify each candidate as (1) *illegal*, (2) *normal operation*, or (3) *fault-revealing*. Do this by executing each candidate and comparing the program's behavior against the operational model (Section 3.2).
- **Reduction.** Partition the *fault-revealing* candidates based on their *violation pattern*: the set of violated properties. Report one candidate from each partition (Section 3.3).

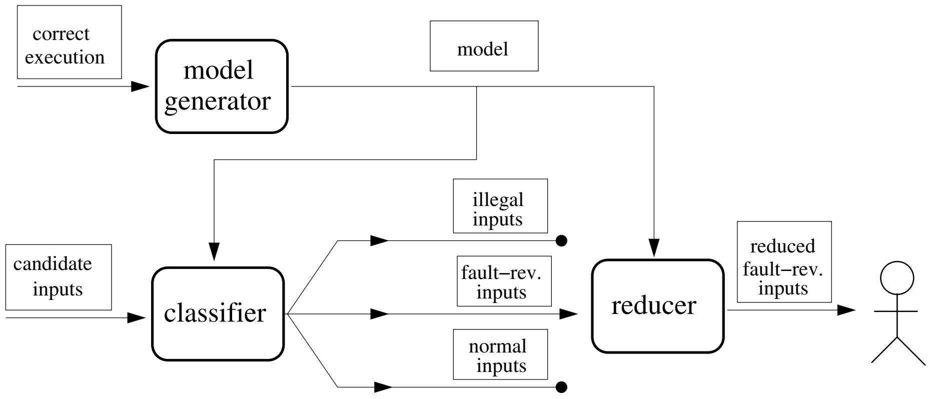


Fig. 4. The input selection technique. Implicit in the diagram is the program under test. Rectangles with rounded corners represent steps in the technique, and rectangles with square corners represent artifacts

Object invariants (hold on entry and exit of all public methods)

$\texttt{max} = \texttt{elems.length}$
$\texttt{elems} \neq \texttt{null}$
$\texttt{max} = 2$
$\texttt{numElems} \geq 0$

Properties that hold on entry to `pop`

$\texttt{elems}[\texttt{max} - \texttt{1}] \geq 0$

Properties that hold on exit from `pop`

$\texttt{elems}[\texttt{max} - \texttt{1}] \geq 0$

Properties that hold on entry to `push`

$\texttt{numElems} \in \{0, 1\}$

Properties that hold on exit from `push`

$\texttt{numElems} \in \{1, 2\}$
$size(\texttt{elems}[]) - 1 \neq \texttt{elems}[\texttt{max} - \texttt{1}]$

Fig. 5. Part of an operational model for `BoundedStack` with respect to an 8-element test suite, generated by the Daikon [11] tool. An operational model reflects particulars of the test suite used to derive it; for example, the last property states that the last element in array `elems` is never equal to its index

3.1 Model Generation

The first step is to generate an operational model of the program. An operational model consists of properties that hold at the boundary of the program's components (e.g., on a public method's entry and exit). Our techniques impose no constraints on the program behavior captured by a model, but they require that every property can be evaluated at runtime.

The Eclat implementation uses operational abstractions generated by the Daikon invariant detector [11]. There are other techniques for generating models of program behavior based on an example use of the program [14, 26, 1, 16]. The models that these techniques generate vary in the kinds of properties they express, from legal sequences of method calls [26] to algebraic specifications of method behavior [16].

Figure 5 shows a simple operational model for `BoundedStack`. In this model, properties are observations about the state of the stack at various program points.

3.2 The Classifier

The classifier takes a candidate input and labels it *illegal*, *normal operation*, or *fault-revealing*. The classifier takes three arguments: a candidate input, the program under test, and an operational model. The classifier runs the program on the candidate input and records which model properties are violated during execution.

A violation means that the candidate input's behavior deviated from previous behavior of the program. Since the previously-seen behavior may be incomplete, such a violation does not necessarily imply faulty behavior. Depending on its violation pattern (the set of violated properties), the classifier labels a candidate input as *illegal*, *normal operation*, or *fault-revealing*. Figure 6 shows the decision table.

Executing an input can result in two kinds of violations: entry or exit violations. Entry violations suggest illegal program inputs, and exit violations suggest improper program behavior. The four possible categories of entry/exit violations are:

– **No entry or exit violations.** This category means that according to the operational model, the program received legal inputs and behaved properly. The technique labels the input *normal operation*.
– **No entry violations, some exit violations.** According to the model, a legal program input led to improper program behavior. The technique labels the input *fault-revealing*.
– **Some entry violations, no exit violations.** The program behaved properly on an illegal input. Since the program behaved properly, the technique labels the input *normal operation*. The program's satisfaction of the exit properties means that it is normal behavior; violation of the entry properties man that it is new behavior not seen in the example correct execution from which the model was generated.
– **Some entry and some exit violations.** The program behaved improperly on an illegal input. The technique labels the input *illegal*.

3.3 The Reducer

Section 3.2 described how an input's violation pattern leads to its classification. Violation patterns also induce a partition on all inputs, with two inputs belonging to the same partition if they violate the same properties. Inputs exhibiting the same pattern of violations are likely to be manifestations of the same faulty program behavior. Consider

Entry violations?	Exit violations?	Classification
no	no	***normal operation***
no	yes	***fault-revealing***
yes	no	**(new)** ***normal operation***
yes	yes	***illegal***

Fig. 6. Decision table for classifying a candidate input, based on the model violations that result from its execution

```
BoundedStack var0 = new BoundedStack();
var0.pop();
```

```
BoundedStack var0 = new BoundedStack();
var0.push(3);
var0.pop();
int var1 = var0.top();
var0.pop();
```

Fig. 7. Two Eclat-generated inputs that reveal the same error in the `pop` method. Both inputs violate the single property `numElems` ≥ 0 on exit from the last `pop`

Figure 7, which contains two fault-revealing inputs. Both inputs violate the same set of properties—namely, the single property `numElems` $\geq$ 0—and they uncover the same error in method `pop`. Presenting only one input will save the user the time to inspect a redundant input.

3.4 Oracle Generation: From Test Input to Test Case

A test engineer's goal is to find errors and to write tests that may find errors in the future. A test consists of an input and an oracle, so providing test inputs, even ones that are likely to be fault-revealing, leaves the test engineer responsible for determining both how the program ought to behave on the input, and how to verify that behavior. This section describes a technique that automatically converts a test input into a test case by proposing an oracle. The human remains the final arbiter of the test suite and should check and/or modify each test case, but the effort can be greatly eased by providing complete test cases rather than partial ones.

The oracle generation technique uses the model described in Section 3.1. Since the properties can be evaluated at run time, they can be converted into assertions and used as test oracles. These oracles check for deviation from previously-observed behavior. In addition to checking behavior, the properties serve as a human-readable explanation of what is being checked, which is important in a test case. Figure 3 shows an example of a test case output by our implementation.

3.5 Classifier-Guided Input Generation

We have presented a technique that selects from a set of candidate inputs a subset likely to reveal faults, and a technique that converts an input into a test case. This section describes a similar methodology to avoid generating illegal inputs in a bottom-up input generation strategy. First we present an unguided strategy for generating inputs, and then we present an enhancement to the strategy that makes use of the classifier from Section 3.2.

We describe input generation in the context of inputs like those in Figure 7, where an input is a sequence of method calls. The last method call is the tested call, with all previous method calls setting up state for the tested call. For example, the second input in Figure 7 has five method calls; the first four are setup, and the fourth one tests the method `pop` via the method call `var0.pop()`.

Unguided Bottom-up Generation. The unguided bottom-up generation strategy maintains a growing pool of values used to construct new inputs. Every value in the pool is accompanied by a code snippet (usually a sequence of method calls) that can be run to construct the value. Each code snippet can be viewed as a test input.

New values are created by combining existing values through method calls. For example, given stack value s and integer value i, the method call s`.isMember(`i`)` creates a new boolean value. Methods that return `void` are treated as producing a new value for the receiver. For example, method call s`.push(`i`)` creates a new stack value.

Bottom-up input generation proceeds in rounds. The pool is initialized with a set of initial values (for example, in Java, a few primitive values and `null`). In each round, new values are created by calling methods and constructors with values from the pool. Each new value is added to the pool and its code is emitted as a test input. The process is repeated any number of times.

Combining Generation and Classification. The unguided generation strategy is likely to produce both interesting inputs and a large number of illegal inputs, since there are no constraints on the arguments passed to method calls. The guided generation technique takes advantage of the classifier to guide the generation process.

As before, input generation proceeds in rounds. For each round:

1. Construct a new set of candidate values (and corresponding inputs) from the existing pool.
2. Classify the new candidate inputs with the classifier.
3. Discard inputs labeled *illegal*, add the values represented by the candidates labeled *normal operation* to the pool, and emit inputs labeled *fault-revealing* (but do not add them to the pool).

Figure 8 illustrates the process (it also adds the oracle generation technique discussed in Section 3.4, to give a complete view of the multiple techniques in a single framework). In the classifier-guided technique, a set of candidate inputs is no longer a required input—it has been replaced by an input generator that uses the classifier to avoid creating illegal inputs.

This enhancement removes *illegal* and *fault-revealing* inputs from the pool as soon as they are discovered, preventing these inputs from being used as building blocks to new method calls (any input that makes such a call would also be classified *illegal*, and is therefore useless to construct).

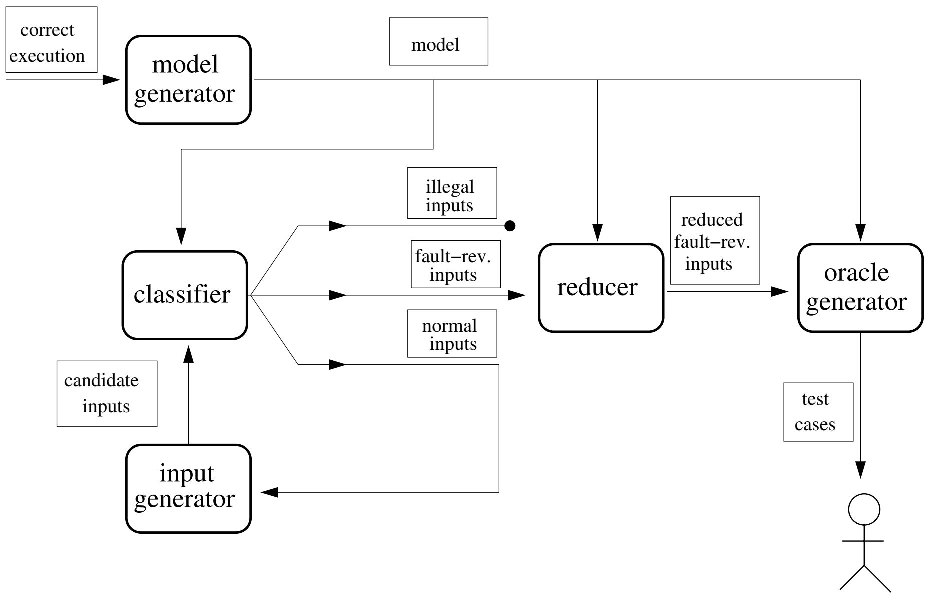


Fig. 8. The input selection technique of Figure 4, augmented with an input generator that uses the classifier to avoid creating illegal inputs, and with an oracle generator that produces test cases from test inputs. This diagram shows all the paper's techniques in a single integrated framework

3.6 Discussion

Applicability. We have presented our test selection technique in the context of an object-oriented programming language. The technique is also applicable in other programming contexts, as long as an operational model can be obtained, the model can be evaluated in the context of new program executions, and the model can be partitioned into entry and exit properties (preconditions and postconditions).

The technique reveals faults that are violations of the model properties. Eclat uses the Daikon invariant detector to infer a model. Daikon infers many kinds of properties about data structures, including heap-based ones, but does not infer, for instance, temporal properties of a program. Thus, one would not expect Eclat to be particularly good at finding faults that have to do with temporal properties.

Integration with manually-written specifications. Our research addresses a testing situation in which the tester has no access to a formal specification, but has a set of correct program executions from which an operational model can be derived. Increasingly, programmers write partial specifications to capture important properties of their software; safety-critical systems, for instance, sometimes contain at least a partial specification of the critical parts of the system. These specifications can be used to generate and classify test inputs. Partial specifications can erroneously classify inputs; for example, an illegal input may be labeled legal because the partially-specified precondition is not strong enough. Our classification technique permits use of manually-written or

Number of rounds	4
Goal number of new invocations per method per round	100
Failed tries after which generation attempts stop for a given method	100
Time limit (generation stops after limit is exceeded)	no limit

Fig. 9. Eclat's default parameters for generating test inputs

mechanically-derived properties, or both. The operational model can be complemented with manually-written specifications that capture important properties not mechanically derived. Conversely, partial specifications can be complemented with inferred properties to improve the input generation and classification process.

4 Implementation: Eclat

We have implemented our input generation, input selection, and oracle generation techniques in Eclat, a tool that automatically creates unit tests for Java classes. Eclat can produce output in text, XML, or a JUnit test suite. Eclat can be used through a command-line interface or as an Eclipse plugin. Eclat is publicly available at `http://pag.csail.mit.edu/eclat/`.

Eclat takes as input a set of classes to test and a program or test suite P that uses the classes. Eclat performs the following steps.

Deriving an operational model. Eclat uses the Daikon dynamic invariant detector [11] to derive a model of the classes' behavior on P; an example of Daikon's output appeared in Figure 5.

Compiling for runtime property checking. We have implemented a run-time-check instrumenter (distributed as part of Daikon at `http://pag.csail.mit.edu/daikon/`). The instrumenter takes the source files of the tested classes and the operational model derived by Daikon. It transforms the sources to check model properties during execution. Instrumentation is transparent: a violation does not alter the behavior of the class. Violated properties are recorded in a log.

Generating candidate inputs. Eclat generates candidate inputs using the classifier-guided, bottom-up generation strategy outlined in Section 3.5. Each round, new inputs are created by calling methods of the tested classes, selecting parameters at random from the pool. For each round, Eclat attempts to create a fixed number of new inputs for a given method using existing values from the pool. After a fixed number of failed attempts, it moves on to the next method. Figure 9 gives Eclat's default parameters. Section 5.6 evaluates Eclat's behavior when varying these parameters.

5 Evaluation

We have run a series of experiments to quantify the effectiveness of our test input generation and selection techniques. Section 5.1 introduces the programs and experimental methodology. Section 5.2 evaluates how well Eclat's selected inputs reveal faults.

Program	versions	suites per version	independent components	classes per component	public methods	NCNB LOC
BoundedStack	1	2	1	1	11	88
DSAA	1	1	9	1.5	110	640
JMLSamples	1	1	25	1.9	221	1392
utilMDE	1	2	1	1	69	1832
RatPoly	97	1	1	4	17	512
Directions	80	2	1	6	42	342

Fig. 10. Subject programs. For programs with multiple versions, numbers are average per version. NCNB LOC means non-comment, non-blank lines of code. These numbers do not include testing code

Section 5.3 measures Eclat's effectiveness when supplied small initial test suites. Sections 5.4–5.6 evaluate the classifier, the reducer, and the classifier-guided input generator individually.

5.1 Subject Programs and Methodology

Figure 10 lists our subject programs. The programs encompass 64 distinct interfaces, and a total of 631 implementations of those interfaces in 75,000 non-comment non-blank lines of code. All subject programs implement modestly-sized libraries designed to support larger programs; thus, unit testing is appropriate for them. All errors are real errors inadvertently introduced by the author(s) of the program.

- BoundedStack is the stack implementation discussed in Section 2. We report separately the results of running Eclat with the 8-test suite, and with the 12-test suite (with the one fault-revealing test removed).
- DSAA is a collection of data structures from an introductory textbook [25]. The author of the classes wrote a small set of example uses of the class: they are not exhaustive tests.
- JMLSamples is a collection of 25 classes that illustrate the use of the JML specification language. It is part of the JML distribution (`www.jmlspecs.org`). The test suites and specifications were written by the authors of the classes.
- utilMDE is a utility package that augments the `java.util` package. We report two results: one running Eclat with the test suite written by the authors of utilMDE, and the other via the unit tests of an unrelated program (Daikon [11]) that uses part of the utilMDE package.
- RatPoly is a set of student solutions to an assignment in MIT class 6.170, Laboratory in Software Engineering. The RatPoly library implements the core of a graphing calculator for polynomials over rational numbers. The course staff provided a test suite to the students as part of the assignment.
- Directions is a different set of student solutions in MIT class 6.170, written by the same students who wrote the RatPoly solutions. The Directions library is used by a MapQuest-like program that outputs directions for traveling from one location to another along Boston-area streets. For this assignment, students wrote their own

test suites. We report separately the results of running Eclat with the student-written suite, and with the suite used by the staff to grade the assignment, which was not provided to the students.

Eclat assumes a correct set of executions. Before running Eclat on BoundedStack and its 12-test suite, which contains one failing test, we removed the failing test.

For RatPoly, we discarded submissions that did not pass the staff test suite, which was provided as part of the assignment. For both RatPoly and Directions, we also discarded submissions for which Eclat generated more than 10 times the average number of fault-revealing inputs. These were solutions so faulty that finding fault-revealing inputs was not challenging, making input selection techniques unnecessary. The numbers in Figure 10 count only versions we kept.

Measurements. We organized our subject programs into nine experiments, each corresponding to using Eclat with a particular subject program and test suite. For a given experiment, we ran Eclat separately on each independent component (for example, we ran Eclat separately on DSAA's nine components: a binary tree, a disjoint set, a treap, an array-backed stack, a list-backed stack, a queue, a red-black tree, a linked list, and a binary heap). Thus, each experiment consisted of potentially many runs of Eclat: one per $\langle$ component, version $\rangle$ pair. For each experiment, we report results that are the average over all runs.

When computing average results for all experiments, we give the same weight to each experiment, regardless of the number of versions or runs of Eclat that the program represents. We do this to avoid over-representing experiments with multiple versions or components.

We wrote formal specifications for all the subject programs (except for JMLSamples, which already had formal specifications written by its authors). We use the specifications to evaluate the classification technique, with the specification representing an ideal classifier. Of course, in the presence of a formal specification our classification technique is not necessary: the specification indicates whether an input is illegal, normal, or fault-revealing. Our techniques are intended for use when formal specifications are not available, as was the case for most of the programs.

Comparison with other tools. JCrasher [9], Jtest [19], and Jov [30] have the same goals as Eclat: to generate random candidate inputs and select potentially fault-revealing ones. We report results from running JCrasher. We tried the other tools, but Jov and Jtest were unusable in many instances (Jov sometimes exited abnormally, and Jtest sometimes failed to terminate).

5.2 Evaluating Eclat's Output

Figure 11 shows how many inputs per run Eclat generated, how many it selected, and how many of those revealed faults. The figure also shows JCrasher's results on the subject programs. The results for JCrasher are the same for experiments that use the same programs with different test suites because JCrasher does not make use of the test suite. We also executed all the inputs against the formal specifications (using `jmlc` [6]). We

	Generated inputs			Selected inputs			JCrasher inputs		
Program	inputs generated	reveal faults	precision	inputs selected	reveal faults	precision	inputs selected	reveal faults	precision
BoundedStack (8-test suite)	806	13	1.6%	3	2	67%	0	0	—
BoundedStack (12-test suite)	1411	22	1.6%	1	1	100%	0	0	—
DSAA	806	0	0%	1.3	0	0%	0.89	0	0%
JMLSamples	396	0.50	0.13%	0.72	0.061	8.4%	0.12	0	0%
utilMDE (test suite)	1787	92	5.1%	18	4	22%	1	0	0%
utilMDE (sample usage)	1774	63	3.6%	18	2	11%	1	0	0%
RatPoly	2862	29	1.0%	1.5	0.65	42%	4	0.13	3.3%
Directions (student suite)	1099	40	3.6%	1.3	0.081	6.4%	1.6	0.025	1.6%
Directions (staff suite)	1099	41	3.8%	0.45	0.079	18%	1.6	0.025	1.6%
average	1338	33	2.3%	5.0	1.1	30%	1.13	0.02	0.92%

Fig. 11. Summary of Eclat's results. The first three numeric columns represent inputs internally generated by Eclat. The next three columns represent inputs reported to the user (after selection and reduction). The last three columns represent inputs selected as fault-revealing by JCrasher. Precision is the percentage of inputs that are fault-revealing. We calculated the average precision by taking the average of the individual experiments; this gives each experiment equal weight, but is slightly different from dividing the average number of fault-revealing inputs by the average number of selected inputs

true label	inputs generated	inputs selected
normal	74%	31%
illegal	24%	38%
fault	2.3%	30%

Fig. 12. True labels of generated and selected inputs. The entries in each column sum to 100% (modulo rounding imprecision). These results represent a total of 440,000 inputs

considered an input fault-revealing if it satisfied all preconditions of the tested method, and the method invocation caused a postcondition violation.

On average, Eclat selected 5.0 inputs per run, and 30% of those revealed a fault. By comparison, JCrasher selected 1.13 inputs per run, and 0.92% of those revealed a fault.

The inputs that Eclat selects are an order of magnitude as likely to reveal faults as the original candidate inputs (30% vs. 2.3%). Figure 12 shows another view of the results: it gives the true label of the generated and selected inputs, i.e., the label assigned by the formal specification. Selection is effective at improving a set of inputs by increasing the ratio of fault-revealing to non-fault-revealing ones.

5.3 Effectiveness on Small Initial Test Suites

Classification depends on a set of correct program executions to derive an approximate model of correct program behavior. This section measures the effect of the initial test suite on Eclat's fault-finding effectiveness. To evaluate the technique's performance on smaller suites, we artificially reduced the set of correct executions used by Eclat to construct an operational model. We compared our previous results with running Eclat using only the first 10% of the original execution trace (which was itself sometimes quite small). The table below shows the results.

	inputs generated	reveal faults	inputs selected	reveal faults
original trace	1338	33	5.0	1.1
10% of trace	1219	29	5.6	1.2

When given a smaller trace, Eclat selected more inputs (5.6 for the small trace, 5.0 for the original trace). Of those, almost the same percentage were fault-revealing.

Generating inputs based on the full-sized trace yields only slightly better results—fewer inputs to inspect, and almost the same number of fault-revealing ones among them. The technique is still effective with an impoverished trace, which makes it useful in the presence of a small test suite that does not cover all aspects of the program's behavior.

The table below shows the percentage of methods covered per test suite, and average number of calls made to each covered method. The number of calls per method covered does not give the whole story, since the distribution is highly non-uniform: in each case (even when test suites exist), a few methods are called many times and most methods are called very few times.

Program	methods covered	calls per method covered
BoundedStack (8-test suite)	82%	8
BoundedStack (12-test suite)	100%	18
DSAA	90%	679
JMLSamples	84%	102
utilMDE (test suite)	46%	13747
utilMDE (sample usage)	1.5%	4
RatPoly	83%	501
Directions (student suite)	85%	330
Directions (staff suite)	85%	3015

For the programs with multiple test suites (BoundedStack, DSAA, and utilMDE), the difference in coverage and number of calls per method is large, but the difference in Eclat's results is smaller.

5.4 Evaluating the Classifier

Every input has two labels, one assigned by Eclat and the true label assigned by the formal specification. Figure 13 shows the proportion of inputs falling into each ⟨Eclat label, true label⟩ category

The last row in Figure 13 shows the *precision* [21, 24] of Eclat's classifier. Precision is the ratio of correct labelings to the total number of labelings:

$$\text{precision} = \frac{\text{inputs correctly labeled as } L}{\text{inputs labeled as } L}$$

true	Eclat label			
label	normal	illegal	fault	**recall**
normal	0.67	0.045	0.030	90%
illegal	0.057	0.17	0.012	24%
fault	0.013	0.0035	0.0058	59%
precision	90%	78%	12%	

Fig. 13. Each entry shows the average proportion of generated inputs with the given Eclat label and true label. The sum of the nine middle entries is 1. The sum of each row in the nine middle entries yields the percentages in the middle column of Figure 12

The last column in Figure 13 shows the *recall* [21, 24] of the classifier. Recall is the ratio of correct labelings to the total number of inputs that belong to the label:

$$\text{recall} = \frac{\text{inputs correctly labeled as } L}{\text{inputs that are actually } L}$$

In summary, the classifier:

– correctly labels the vast majority of inputs as non-fault-revealing (90% precision, 90% recall for normal inputs),
– recognizes most fault-revealing inputs (59% precision for fault-revealing inputs), but
– labels fault-revealing many inputs that are not (12% precision for fault-revealing inputs).

The degree to which the technique overclassifies normal inputs as illegal depends on the accuracy with which the operational model captures the legality of the program's inputs. An operational model that is out of sync with the true input space of the program can indicate a poor test suite. A good example of this is BoundedStack. This interface permits arbitrary sequences of method calls with arbitrary parameters, so it is impossible to produce an illegal input, but the technique classifies many inputs as such, due to the test suite's poor coverage. When a test engineer inspects an input that is incorrectly classified as fault-revealing, the engineer is likely to find weaknesses in the test suite, permitting the engineer to improve it.

Identifying new behavior. Our technique classifies inputs into one of three labels: *illegal*, *normal operation* and *fault-revealing*. As shown in Figure 6, there are two kinds of normal inputs: those that violate no model properties, and those that violate some preconditions but no postconditions. The latter, called *new* inputs, are inputs that diverge from the original test suite, but the properties they violate are not considered indicative of faults; instead they are considered indicative of an overconstrained model. We experimented with outputting the *new* inputs for user inspection along with the fault-revealing ones, but we found that new behaviors were no more effective in revealing faults than normal behaviors that violate no properties. However, distinguishing new behaviors from old ones might help the programmer improve a test suite's coverage by suggesting normal program operation not already covered by the suite.

5.5 Evaluating the Reducer

The reducer takes the inputs labeled *fault-revealing*, and retains a representative subset. The table below summarizes its behavior. The first numeric column shows the average distribution of all inputs that the classifier labeled *fault-revealing* (the input to the reducer). The next column shows the distribution of inputs selected (the output of the reducer). Each column sums to 100%, modulo rounding imprecision.

true label	inputs labeled as fault by classifier	inputs selected (reduced)
normal	63%	31%
illegal	25%	38%
fault	12%	30%

The reduction step increases the percentage of fault-revealing inputs from 12% to 30%. For these programs (and, we suspect, for programs in general), fault-revealing program behavior is more difficult to produce than illegal or normal behavior, and thus more difficult to produce repeatedly by different inputs. This makes fault-revealing inputs less reducible than other inputs, because there are fewer inputs per partition, resulting in an increased proportion of selected fault-revealing inputs.

5.6 Evaluating the Input Generator

Classifier-guided Input Generation. Section 3.5 describes the use of the classifier in a bottom-up input generation strategy in which only inputs classified as *normal operation* are added to the growing pool of inputs. The first line in Figure 14 shows the results of this strategy (Eclat's default) for the formally-specified programs (this line repeats the averages from Figure 11). The second line shows the result of running Eclat using unguided generation: all inputs from previous rounds are added to the pool regardless of their classification.

Unguided generation leads to a larger number of inputs generated. The reason is that the pool has a larger number of building blocks to create new inputs from. Despite the larger number of inputs generated, fewer of those inputs are fault revealing. This is reflected in the results: with the unguided generation strategy, Eclat reports a larger number of inputs and yet fewer inputs are fault-revealing.

We can gain insight into this difference by looking back at Figure 12, which shows that the input selection technique selects not only more *fault-revealing* inputs, but also more *illegal* inputs. Eclat is most effective at correctly classifying normal inputs, but less so for illegal ones. When we remove the classifier from the generation process, the number of illegal inputs among candidate inputs increases, and Eclat selects more of them as fault-revealing, which decreases the tool's precision. Constraining the building blocks used by the generator to inputs classified as *normal operation* reduces these false positives.

Generation Parameters. This section evaluates Eclat's output under varying parameters. We varied two parameters:

	inputs generated	reveal faults	inputs selected	reveal faults
classifier-guided generation	1338	33	5.0	1.1
unguided bottom-up generation	3217	17	5.3	0.80

Fig. 14. Comparison of unguided and enhanced bottom-up generation. The first line summarizes the results for classifier-guided generation (averages reproduced from Figure 11). The second line uses unguided input generation

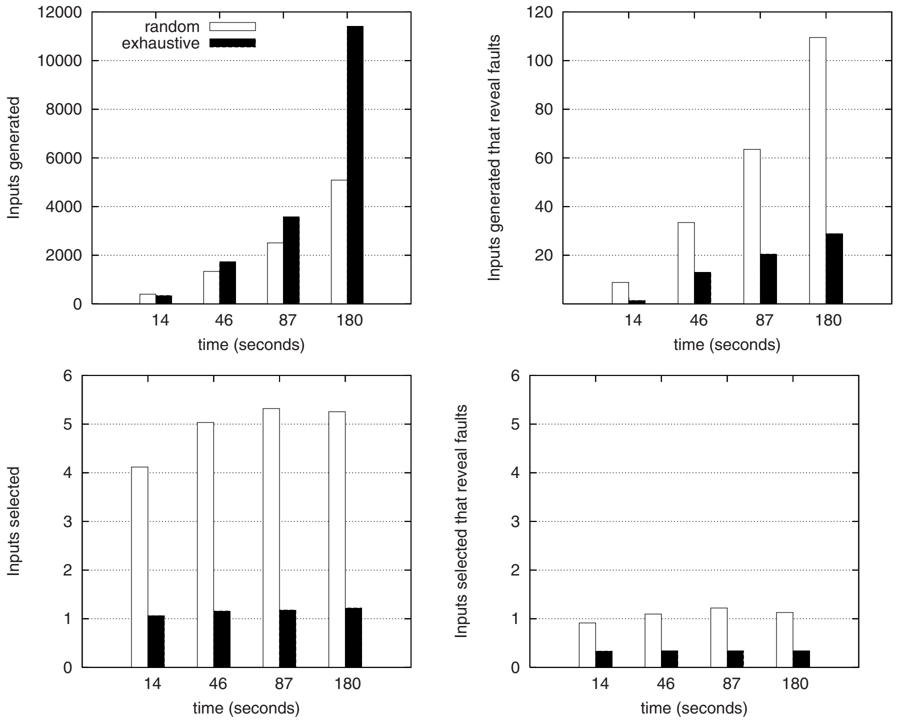


Fig. 15. Number of inputs generated and selected by Eclat, when varying the number of rounds and the generation strategy. The white bars are the results of running Eclat using random generation. The four data points are for the end-to-end time Eclat takes doing 2, 4, 6, and 8 rounds of random generation. The black bars are the results of running Eclat using exhaustive generation. The times shown are averages over all experiments

- The number of rounds of bottom-up generation. Eclat's default is 4 rounds; we also ran the experiments using 2, 6, and 8 rounds of generation.
- The number of new inputs generated per round. Eclat's default is to randomly generate 100 new inputs per method per round. To compare this approach against a more systematic approach, we added exhaustive generation to Eclat: for each round, it exhaustively generates all new inputs that are possible to generate given the current pool of values. To compare this approach against random generation, we mea-

sured how random and exhaustive generation performed given the same amount of time. We measured the time that Eclat spent generating, classifying and reducing inputs using random generation for a given number of rounds, and we ran Eclat again, using exhaustive generation and setting a time limit equal to the time spent by random generation.

Figure 15 shows the results for the eight possible combinations of parameter variations described above. Given the same amount of time, random generation generates fewer candidate inputs (upper-left plot). At every attempt to generate a new input for a method, Eclat's random generation algorithm randomly chooses a set of parameters, and then checks to see if the input has already been generated. This adds two costs to random generation: the cost of comparing a newly-generated random input for membership in the set of existing inputs, and the wasted cost of generating an input that is already in the pool. Exhaustive generation, on the other hand, never re-generates an already-existing input.

Despite creating fewer candidate inputs, random generation produces better-quality candidates—candidates that are fault-revealing (upper-right plot). Exhaustive generation creates many inputs that exercise the class in ways that are indistinguishable for the purpose of fault detection. Random generation produces a more diverse collection of inputs and more fault-revealing inputs than exhaustive generation (bottom plots). In future work, we plan to investigate exhaustive generation combined with techniques for avoiding generation of duplicate inputs [28, 29].

6 Related Work

The most closely related work to ours is the Jov [30] and JCrasher [9] tools, which share the goal of selecting, from a randomly-generated set of candidate inputs, a set most likely to be useful. This reduces the number of test inputs a human must examine.

Our research was inspired by Jov [30]. Jov builds on earlier work [15] that identified a test as a potentially valuable addition to a test suite if the test violates an operational abstraction built from the suite: the test represents some combination of values that differs from all tests currently in the suite. (The DIDUCE tool [14] takes a similar approach, though with the goal of identifying bugs at run time rather than improving test suites: a property that has held for part of a run, but is later violated, is suggestive of an error.) The Jov tool uses the operational abstraction not just to select tests, but also to guide test generation, by iterated use of the Jtest tool [19]. Jov also differs from the previous, automated work on test selection [15] by placing it in a loop with human interaction and iterating as many times as desired:

1. Create an operational model (invariants) from a test suite.
2. Generate test inputs that violate the invariants.
3. A human selects some of the generated tests and adds them to the test suite.

Often, overconstrained preconditions rendered Jtest incapable of producing any outputs, so Xie and Notkin report on the effectiveness of Jov after eliminating all preconditions from the operational model generated in step 1. Essentially, this permitted Jtest to generate any input that violates the postconditions (including many illegal ones), not just inputs similar to the ones in the original test suite. However, the user gets no help in recognizing such illegal inputs. In fact, the majority of errors that Jov finds [30] are illegal inputs and precondition violations, not true errors [27].

Our work extends that of Xie and Notkin in several ways. Our technique explicitly addresses the imperfect nature of a derived operational model. Our technique explicitly distinguishes between illegal and fault-revealing inputs. Our technique is more automated: it requires only one round of examination by a human, rather than multiple rounds. Our technique uses operational abstractions in a different way to direct test input generation. Our implementation is more robust and faster; Eclat takes less than two minutes for a class that took Jov over 10 minutes to process, primarily because the Jtest tool is so slow. We have performed a more extensive experimental evaluation (631 classes rather than 12). Even though we count only actual errors, not illegal inputs, our approach outperforms the previous one.

JCrasher [9], like Eclat, generates a large number of random inputs and selects a small number of potentially fault-revealing ones. An input is considered potentially fault-revealing if it throws an undeclared runtime exception. Inputs are grouped (reduced) based on the contents of the call-stack when the exception is thrown. JCrasher and Eclat have similar underlying generation techniques but different models of correct program behavior, which leads to different classification and reduction techniques. JCrasher's model takes into account only exceptional behavior, and Eclat augments the model with operational behavior, which accounts for its greater effectiveness in uncovering faults.

6.1 Future Work

Future work on this research centers around two themes.

– **Input generation.** While it may not help in establishing the reliability of a program, random testing seems to be remarkably effective in exposing errors and may be as effective as more formally founded techniques [10, 13]. However, it is primarily useful when all inputs are legal, or when a specification of valid inputs is available. Therefore, techniques that make it more effective are valuable contributions. Our technique could be combined with any technique for generating tests [8, 4], in order to filter the tests before being presented to a user. Our technique is attractive because it does not require a human-written formal specification; when one is present, much more powerful testing methodologies are possible [2, 7].
– **Input classification.** Eclat's reduction step clusters test inputs in order to reduce their number, and JCrasher has a similar step. Several researchers have used machine learning to classify program executions as either correct or faulty [20, 5, 3]. It would be interesting to apply such techniques in order to further improve Eclat.

7 Conclusion

We have presented an input selection technique that incorporates a classifier and a reducer, both of which make use of a model of correct program operation. We have combined our input selection technique with two other techniques. One technique uses the classifier to guide input generation towards legal inputs, which improves the efficiency of the input search space by pruning illegal sequences of methods calls as early as they are encountered. The other additional technique uses the operational model to produce oracles for the selected test inputs, which converts the test inputs into full-fledged test cases. Together, these techniques result in an effective test generation and selection methodology.

We have implemented the methodology in Eclat, a tool for Java unit testing, and demonstrated its effectiveness in producing fault-revealing test inputs. The input generation technique creates legal, fault-revealing candidate inputs for the methods in our subject programs, and the input selection technique selects inputs that are an order of magnitude as likely to reveal faults as the candidate inputs. The methodology reveals real, previously unknown errors in the subject programs. When the test inputs fail to reveal faults, the user is not heavily inconvenienced, because only a few inputs are selected.

References

[1] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, Portland, Oregon, Jan. 16–18, 2002.

[2] M. J. Balcer, W. M. Hasling, and T. J. Ostrand. Automatic generation of test scripts from formal test specifications. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Testing, Analysis, and Verification (TAV3)*, pages 210–218, Dec. 1989.

[3] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *ISSTA 2004, Proceedings of the 2004 International Symposium on Software Testing and Analysis*, pages 195–205, Boston, MA, USA, July 12–14, 2004.

[4] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 123–133, Rome, Italy, July 22–24, 2002.

[5] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE'04, Proceedings of the 26th International Conference on Software Engineering*, pages 480–490, Edinburgh, Scotland, May 26–28, 2004.

[6] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, Trondheim, Norway, June 5–7, 2003.

[7] J. Chang and D. J. Richardson. Structural specification-based testing: Automated support and experimental evaluation. In *Proceedings of the 7th European Software Engineering Conference and the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 285–302, Toulouse, France, Sept. 6–9, 1999.

[8] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP '00, Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*, pages 268–279, Montreal, Canada, Sept. 18–20, 2000.

[9] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1117, Sept. 2004.

[10] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4):438–444, July 1984.

[11] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, Feb. 2001. A previous version appeared in *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

[12] Foundations of Software Engineering group, Microsoft Research. *Documentation for AsmL 2*, 2003. `http://research.microsoft.com/fse/asml`.

[13] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, Dec. 1990.

[14] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE'02, Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, Orlando, Florida, May 22–24, 2002.

[15] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *ICSE'03, Proceedings of the 25th International Conference on Software Engineering*, pages 60–71, Portland, Oregon, May 6–8, 2003.

[16] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *ECOOP 2003 — Object-Oriented Programming, 17th European Conference*, pages 431–456, Darmstadt, Germany, July 23–25, 2003.

[17] B. Korel. Automated test data generation for programs with procedures. In *Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pages 209–215. ACM Press, 1996.

[18] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[19] Parasoft Corporation. *Jtest version 4.5*. `http://www.parasoft.com/`.

[20] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *ICSE'03, Proceedings of the 25th International Conference on Software Engineering*, pages 465–475, Portland, Oregon, May 6–8, 2003.

[21] G. Salton. *Automatic Information Organization and Retrieval.* McGraw-Hill, 1968.

[22] D. Stotts, M. Lindsey, and A. Antley. An informal formal method for systematic JUnit test case generation. In *Proceedings of 2nd XP Universe and 1st Agile Universe Conference (XP/Agile Universe)*, pages 131–143, Chicago, IL, USA, Aug. 4–7, 2002.

[23] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In *Proceedings of the 13th Annual International Conference on Automated Software Engineering (ASE'98)*, pages 285–288, Honolulu, Hawaii, Oct. 14–16, 1998.

[24] C. J. van Rijsbergen. *Information Retrieval.* Butterworths, London, second edition, 1979.

[25] M. A. Weiss. *Data Structures and Algorithm Analysis in Java.* Addison Wesley Longman, 1999.

[26] J. Whaley, M. Martin, and M. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 218–228, Rome, Italy, July 22–24, 2002.

[27] T. Xie. Personal communication, Aug. 2003.

[28] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *ASE 2004: Proceedings of the 20th Annual International Conference on Automated Software Engineering*, pages 196–205, Linz, Australia, Nov. 9–11, 2004.

[29] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 365–381, Edinburgh, UK, Apr. 4–8, 2005.

[30] T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In *ASE 2003: Proceedings of the 18th Annual International Conference on Automated Software Engineering*, pages 40–48, Montreal, Canada, Oct. 8–10, 2003.

Lightweight Defect Localization for Java

Valentin Dallmeier, Christian Lindig, and Andreas Zeller

Saarland University, Saarbrücken, Germany
{dallmeier, lindig, zeller}@cs.uni-sb.de

Abstract. A common method to localize defects is to compare the *coverage* of passing and failing program runs: A method executed only in failing runs, for instance, is likely to point to the defect. However, some failures, occur only after a specific *sequence* of method calls, such as multiple deallocations of the same resource. Such sequences can be collected from arbitrary Java programs at low cost; comparing object-specific sequences predicts defects better than simply comparing coverage. In a controlled experiment, our technique pinpointed the defective class in 39% of all test runs.

1 Introduction

Of all debugging activities, locating the defect that causes the failure is by far the most time-consuming. To assist the programmer in this task, various automatic methods rank the program statements by the *likelihood* that they contain the defect. One of the most lightweight methods to obtain such a likelihood is to compare the *coverage* of *passing* and *failing* program runs: A method executed only in failing runs, but never in passing runs, is correlated with failure and thus likely to point to the defect.

Some failures, though, come to be only through a *sequence* of method calls, tied to a *specific object.* As an example, consider streams in Java: If a stream is not explicitly closed after usage, its destructor will eventually do so. However, if too many files are left open before the garbage collector destroys the unused streams, file handles will run out, and a failure occurs. This problem is indicated by a sequence of method calls: if the last access (say, `read()`) is followed by `finalize()` (but not `close()`), we have a defect.

In this paper, we explore comparing call sequences between program runs for defect localizaton. Specifically, we explore three questions:

1. **Are *sequences of method calls* better defect indicators than single calls?** In any Java stream, calls to `read()` and `finalize()` are common; but the sequence of these two indicates a missing `close()` and hence a defect.
2. **Do method calls indicate defects more precisely when collected *per object,* rather than globally?** The sequence of `read()` and `finalize()` is only defect-revealing when the calls pertain to the same object.
3. **Do missing (or extra) method calls indicate defects in the callee—*or in the caller?*** For any Java stream, a missing `close()` indicates a defect in the caller.

Generalizing to arbitrary method calls and arbitrary defects, we have set up a tool that instruments a given Java program such that sequences of method calls are collected on a per-object basis. Using this tool, we have conducted two experiments that answer the above questions. In short, it turns out that (1) sequences predict defects better than simply comparing coverage, (2) per-object sequences are better predictors than global sequences, and (3) the caller is more likely to be defective than the callee. Furthermore, the approach is lightweight in the sense that the performance is comparable to coverage-based approaches. All these constitute the contribution of this paper.

2 How Call Sequences Indicate Defects

Let us start with a phenomenological walkthrough and take a look at the AspectJ compiler—more precisely, at its bug #30168. This bug manifests itself as follows: Compiling the AspectJ program in Fig. 1 produces illegal bytecode that causes the virtual machine to crash (run $r_{\times}$):

```
$ ajc Test3.aj
$ java test.Test3
test.Test3@b8df17.x

Unexpected Signal : 11 occurred at PC=0xFA415A00
Function name=(N/A)
Library=(N/A)
...
Please report this error at http://java.sun.com/...
$ _
```

As the bug not only affects execution of the Java program per se, but crashes the virtual machine completely, there is no hint to the origin of the problem like for example a stack trace. As the AspectJ compiler has 2,929 classes, finding the location of the defect is a nontrivial task. To ease the task, though, we can focus on *differences* in the program execution, in particular the difference between a passing run (producing valid Java bytecode) and the failing run in question. Since the outcome of passing and failing runs is different, chances are that earlier differences in the program runs are related to the defect. For the AspectJ example in Figure 1, we can easily identify a passing run—commenting out Line 32, for instance, makes AspectJ work just fine (run $r_{\checkmark}$).

Since capturing and comparing entire runs is costly, researchers have turned to *abstractions* that summarize essential properties of a program run. One such abstraction is *coverage*—that is, the pieces of code that were executed in a run. Indeed, comparing the coverage of $r_{\checkmark}$ and $r_{\times}$ reveals a number of differences. The method `getThisJoinPointVar()` of the class `BcelShadow`, for instance, is only called in $r_{\times}$, but not in $r_{\checkmark}$, which makes `BcelShadow.getThisJoinPointVar()` a potential candidate for causing the failure.

Unfortunately, this hypothesis is wrong. In our AspectJ problem, the developers have eventually chosen to fix the bug in another class; therefore, `BcelShadow` is not the location of the defect. In fact, none of the methods that are called *only* within $r_{\times}$ contain the defect.

However, it may well be that the failure is caused not by a single method call, but rather by a *sequence of method calls* that occurs only in the failing run $r_{\times}$. Such se-

```
package test;
import org.aspectj.lang.*;
import org.aspectj.lang.reflect.*;

public class Test3 {
   public static void main(String[] args) throws Exception {
      Test3 a = new Test3();
      a.foo(-3);
   }
   public void foo(int i) {
      this.x=i;
   }
   int x;
}

aspect Log {
   pointcut assign(Object newval, Object targ):
      set(* test..*)  && args(newval) && target(targ);

   before(Object newval, Object targ): assign(newval,targ) {
      Signature sign = thisJoinPoint.getSignature();
      System.out.println(targ.toString() + "." + sign.getName() +
                         ":=" + newval);
   }

   pointcut tracedCall():
      call(* test..*(..)) && !within(Log);

   after() returning (Object o):  tracedCall() {
      // Works if you comment out either of these two lines
      thisJoinPoint.getSignature();
      System.out.println(thisJoinPoint);
   }
}
```

Fig. 1. This AspectJ program causes the Java virtual machine to crash

quences can be collected for specific objects. This sequence, for instance, summarizes method calls initiated by an instance of `ThisJoinPointVisitor` in $r_{\times}$:

$$\left\langle \begin{array}{l} \texttt{ThisJoinPointVisitor.isRef(),} \\ \texttt{ThisJoinPointVisitor.canTreatAsStatic(),} \\ \texttt{MethodDeclaration.traverse(),} \\ \texttt{ThisJoinPointVisitor.isRef(),} \\ \texttt{ThisJoinPointVisitor.isRef()} \end{array} \right\rangle$$

This sequence of calls does not occur in $r_{\checkmark}$—in other words, only in $r_{\times}$ did an object of the `ThisJoinPointVisitor` class call these five methods in succession. This difference in the `ThisJoinPointVisitor` behavior is correlated with failure and thus makes `ThisJoinPointVisitor` a class that is more likely to contain the defect. And indeed, it turns out that AspectJ bug #30168 was eventually fixed in `ThisJoinPointVisitor`. Thus, while a difference in coverage may not point to a defect, a difference in call sequences may well.

Comparing two runs usually yields more than one differing sequence. In our case ($r_{\checkmark}$ vs. $r_{\times}$), we obtain a total of 556 differing sequences of length 5. We can determine the originating class for each of these sequences, assign a weight to each sequence, and rank the classes such that those with the most important sequences are at the top. In this ranking, the `ThisJoinPointVisitor` class is at position 6 out of 542 executed classes—meaning that the programmer, starting at the top, has to examine only 1.1% of the executed classes or 3.2% of the executed code (0.2% of all classes or 0.8% of the entire code) in order to find the defect. (In comparison, if we had compared only the method coverage of $r_{✓}$ and $r_{✗}$, we would have discovered no difference and hence no indication that the defect is located in `ThisJoinPointVisitor`.)

While such anecdotal evidence is nice, we had to evaluate our approach more thoroughly. In the remainder of this paper, we first describe in detail how we collect sequences of method calls (Section 3), and how we compare them to detect likely defects (Section 4). In Section 5, we describe our experiments with the NanoXML parser and AspectJ; the results support our initial claims. Section 7 discusses related work and Section 8 closes with conclusion and consequences.

3 Summarizing Call Sequences

Over its lifetime, an object may receive and initiate millions of method calls. How do we capture and summarize these to characterize normal behavior? These are the highlights of our approach:

- Recording a trace of all calls per object quickly becomes unmanageable and is a problem in itself (Reiss and Renieris, 2001). Rather than recording the full trace, we abstract from it by *sliding a window* over the trace and remembering only the observed substrings of calls in a *call-sequence set*.
- Collecting a sequence set per object is still problematic, as an application may instantiate huge numbers of objects. We therefore *aggregate* sequence sets into *one set per class*, which thus characterizes the behavior of the class.
- An object receives and initiates method calls. The trace of *incoming* (received) calls tells us how an object is *used* by its clients. The trace of *outgoing* (initiated) calls tells us how an object is *implemented.* We consider both types of traces for fault localization.
- We keep the *overhead* for collecting and analyzing traces as low as possible. Overall, the overhead is comparable to measuring coverage—and thus affordable even in the field.

The following sections describe these techniques in detail.

3.1 From Traces to Call Sequences

A *trace* is an observation of events over the lifetime of an objects, class, or program. In order to capture an object's behavior, we can record the calls it initiates or receives. For realistic runs, these traces are very large. Our approach therefore uses a more abstract representation of an object's behaviour. Instead of investigating whole traces, we remember only *characteristic sequences* of calls. This abstraction of a trace works equally well for a trace of initiated or received calls, or any other trace, which is why we talk about traces in general.

When we slide a window over a trace, the contents of the window characterize the trace—as demonstrated in Fig. 2. The observed window contents form a set of short sequences. The wider the window, the more precise the characteristic set will be.

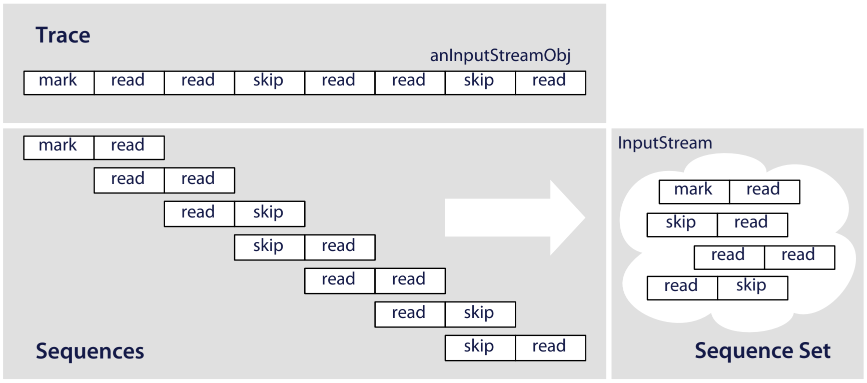


Fig. 2. The call trace of an object is abstracted to a *call-sequence set* using a sliding window

Formally, a trace S is a string of calls: $\langle m_1, \ldots, m_n \rangle$. When the window is k calls wide, the set $P(S,k)$ of observed windows are the k-long substrings of S: $P(S,k) = \{w \mid w \text{ is a substring of } S \wedge |w| = k\}$. For example, consider a window of size $k = 2$ slid over S and the resulting set of sequences $P(S,2)$:

$$S = \langle abcabcdc \rangle \qquad P(S,2) = \{\langle ab \rangle, \langle bc \rangle, \langle ca \rangle, \langle cd \rangle, \langle dc \rangle\}$$

Obviously different traces may lead to the same set: for $T = \langle abcdcdca \rangle$, we have $P(T,2) = P(S,2)$. Hence, going from a trace to its sequence set entails a loss of information. The equivalence of traces is controlled by the window size k, which models the context sensitivity of our approach: in the above example a window size $k \geq 3$ leads to different sets $P(S,k)$ and $P(T,k)$. In the remainder of the paper, we use $P(T)$ to denote the sequence set computed from T, not mentioning the fixed k explicitly.

Note that two calls that are next to each other in a sequence may have been far apart in time: between the two points in time when the object received or initiated the calls, other objects may have been active.

If a trace has less entries than the window size, the missing entries are filled up with dummy invocations that can be distinguished from regular entries. Thus, every sequence set for a trace contains at least one entry.

The size of a sequence set may grow exponentially in theory: With n distinct methods, n^k different sequences of length k exist. In practice, sequence sets are small because method calls are induced by code, which is static. Hence, loops in the code lead to reoccuring sequences that make sequence sets a useful and compact abstraction—one could also consider them an *invariant* of program behavior.

Much of the versatility of sequence sets is due to their set nature: this makes it easy to aggregate and compare sequence set, unlike tree- or graph-based representations (Reiss and Renieris, 2001; Ammons et al., 2002).

3.2 From Objects to Classes

Collecting one sequence set per object raises an important issue: In a program with millions of objects, we will quickly run out of memory . As an alternative, one could think about tracing calls at the *class level* to derive one sequence set per class. In an implementation of such a trace, an object adds an entry to the trace of its class every time it receives (or initiates) a call. Sliding a window over this trace results in a sequence set that characterizes the class's behavior.

As an example of sequence sets aggregated at class level, consider the traces X and Y of two objects. Both objects are *live at the same time* and because we are collecting one trace S per class, their calls interleave in this trace:

$$\begin{aligned} X &= \langle \;\; a \;\; b \;\; c \;\; d \quad dc \rangle \\ Y &= \langle a \;\; a \;\; b \;\; c \;\; ab \quad\; \rangle \\ S &= \langle aaabbccdabdc \rangle \\ P(S,2) &= \{\langle aa\rangle, \langle ab\rangle, \langle bb\rangle, \langle bc\rangle, \langle cc\rangle, \langle cd\rangle, \langle da\rangle, \langle bd\rangle, \langle dc\rangle\} \end{aligned}$$

The resulting sequence set $P(S,2)$ characterizes the behavior of the class—somewhat. The set contains sequences like $\langle da\rangle$ or $\langle bb\rangle$ that we never observed at the object level. How objects interleave has a strong impact on the class trace S, and consequently on its sequence set. This becomes even more obvious when a class instantiates many objects and when their interleaving becomes non-deterministic, as in the presence of threads.

We therefore use a better alternative: We trace objects *individually,* but rather than aggregating their traces, we aggregate their *sequence sets.* Previously, we collected all calls into one trace and computed its sequence set. Now, we have individual traces, but combine their sequence sets into one set per class. The result $P(X,2) \cup P(Y,2)$ is more faithful to the traces we actually observed—$\langle bb\rangle$ and $\langle da\rangle$ are no longer elements of the sequence set:

$$\begin{aligned} P(X,2) &= \{\langle ab\rangle, \langle bc\rangle, \langle cd\rangle, \langle dd\rangle, \langle dc\rangle\} \\ P(Y,2) &= \{\langle ab\rangle, \langle bc\rangle, \langle ca\rangle, \langle aa\rangle\} \\ P(X,2) \cup P(Y,2) &= \{\langle aa\rangle, \langle ab\rangle, \langle bc\rangle, \langle cd\rangle, \langle dd\rangle, \langle dc\rangle, \langle ca\rangle\} \end{aligned}$$

The sequence set of a class is the union of the sequence sets of its objects. It characterizes the behavior of the class and is our measure when comparing classes in passing and failing runs: we simply compare their sequence sets.

3.3 Incoming vs. Outgoing Calls

Any object receives incoming and initiates outgoing method calls. Their traces tell us how the object is used by its clients and how it is implemented, respectively. Both kinds of traces can be used to detect control flow differences between a passing and a failing run. However, they differ in their ability to relate those differences to defects.

As an example, consider object `aQueue` in Fig. 3. The queue receives calls like `enqueue()` to add an element, and `dequeue()` to remove it. These are *incoming* calls to object `aQueue`.

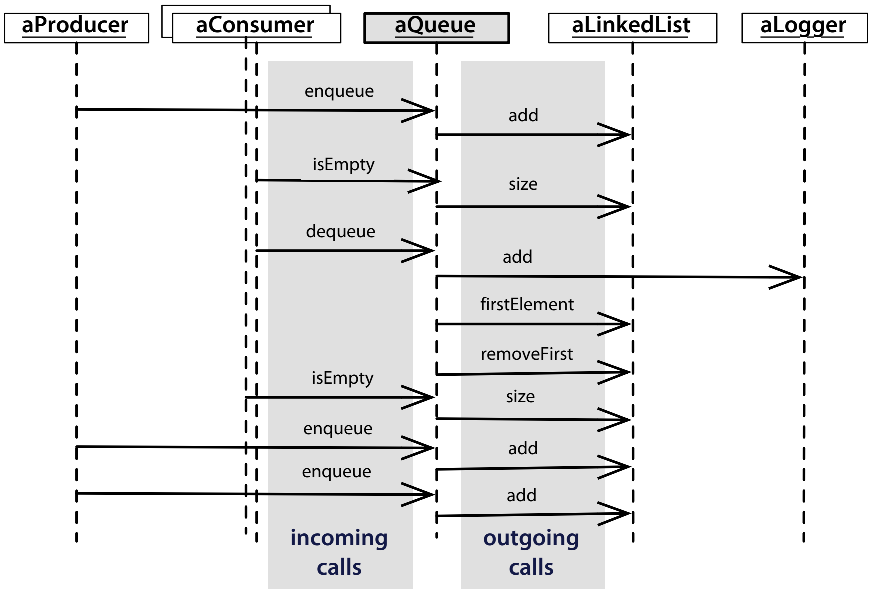


Fig. 3. Traces of *incoming calls* (left) and *outgoing calls* (right) for object `aQueue`

To implement these methods, the queue object uses another object `aLinkedList`. It calls `add()` to add an element at the end of the linked list, `firstElement()` to obtain the first element, and `removeFirst()` to remove it from the list. These calls are *outgoing* calls of object `aQueue`.

Incoming Calls. Inspired by the work of Ammons et al. (2002), we first examined *incoming* calls. The technique of Ammons et al. observes clients that call into a part of the X11 API and learns automatically a finite-state automaton that describes how the API is used correctly by a client: for example, a client must call `open()` before it may call `write()`. Such an automaton is an invariant of the API; it can be used to detect non-conforming clients.

By tracing incoming calls, we can also learn this invariant and represent it as a sequence set: each object traces the calls it receives. Since we know the class `Queue` of the receiving object, we have to remember in a sequence only the names of the invoked methods (and their signatures, to resolve overloading). In our example, the trace of incoming calls for the `aQueue` object is

$$\langle \texttt{enqueue()}, \texttt{isEmpty()}, \ldots, \texttt{enqueue()}, \texttt{enqueue()} \rangle \ .$$

As discussed in Section 3.2, sequence sets of individual objects are aggregated into one sequence set per class. After training with several passing runs, we can detect when a class receives calls that do not match a learned sequence set.

Learning class invariants from incoming calls is appealing for at least two reasons: First, the number of methods an object can receive is restricted by its class. We thus can expect small traces and may even fine-tune the window size in relation to its number of methods. Second, class invariants could be learned across several applications that use the class, not just one.

Outgoing Calls. In our setting, incoming calls show a major weakness: When we detect a non-conforming usage of a class, it is difficult to identify the responsible client. For example, let us assume we observe a new sequence of incoming calls like ⟨`dequeue(),dequeue(),dequeue()`⟩. This sequence could indicate a problem because a consumer should check for an empty queue using `isEmpty()` before attempting a `dequeue()`. The sequence could also be harmless, for instance, when the `dequeue()` calls stem from different objects. In any case, it is not the queue object which is responsible for the new sequence, but the objects that initiated the `dequeue()` calls. Consequently, we turned from incoming to *outgoing* calls, which summarize the method calls initiated by an object. For `aQueue`, these are:

⟨`LinkedList.add(),LinkedList.size(),Logger.add(),...`⟩

Because an object may call objects from several classes, method names are no longer unique—witness the different calls to `add`. We therefore remember the class *and* method name in a trace. Again, we build one trace per object and aggregate the traces of individual queue objects into one sequence per class, which represents its behavior.

When we detect a sequence of outgoing calls that is not in a learned sequence set, we know where to look for the reason: the `Queue` class. Unlike a trace of incoming calls, the trace of outgoing calls can guide the programmer to the defect.

3.4 Collecting Traces

We trace a Java program using a combination of off-line and on-line methods. Before the program is executed, we instrument its bytecode for tracing. While it is running, the program collects traces, computes the corresponding sequence sets, and emits them in XML format before it quits; analyzing sequence sets takes place offline.

For program instrumentation, we use the Bytecode Engineering Library (BCEL, Dahm (1999)). This requires just the program's class files and works with any Java virtual machine. We thus can instrument any Java application, regardless of whether its source code is available. While this is not a typical scenario for debugging, it allows us to instrument the SPEC JVM 98 benchmark, or indeed any third-party code.

Instrumentation of a class rewrites all call sites and the start of every non-static method. The code injected at call sites is needed to determine the caller of a method invocation: because of dynamic binding, a caller cannot statically know the exact class of the method called, and a method (without inspecting the stack) does not know the calling class.

A call is rewritten such that, before a call occurs, the caller's class and instance identifers are written to a thread-local variable from where they are read by code added to the prolog of the called method (the callee). The callee finally enters the actual call to the trace of the caller (when tracing outgoing calls) or callee (when tracing incoming calls).

```
class Caller extends Object {
  public void call() {
    Callee c;
    ...
    Tracer.storeCaller(this.id);
    c.method();
    ...
  }
}
```

```
class Callee extends Object {
  public void method() {
    Tracer.addCall(⟨id for Callee.method>⟩);
    ...
  }
}
```

Fig. 4. Instrumentation of caller and callee to capture outgoing calls

Each object has its own trace of incoming or outgoing calls, but the trace is not stored within the object. Instead, trace data associated with an object is stored in global hash tables. Since Java's `Object.hashCode()` method is unreliable for object identification, each object creates a unique integer for identification in its constructor. Keeping trace data outside of objects has the advantage that they can be accessed by foreign objects, which is essential for outgoing calls.

For an incoming call, the callee simply adds its name and signature to *its own* trace. But for an outgoing call, the callee must add its name, signature, and class to the trace of the *caller*. To do so, it needs to access the caller's trace using the caller's id.

Fig. 4 presents a small example illustrating instrumentation for tracing outgoing calls. (For the sake of readability, we provide Java code instead of byte code.) Statements added during the instrumentation are shown in bold face. Prior to the invocation of `Callee.method()` in method `Caller.call()`, the id of the caller is stored in the `Tracer`. At the very start of `Callee.method()`, `Tracer.addCall()` adds the method id of `Callee.message()` to the trace of the calling object—the one which was previously stored in the `Tracer`. Hence, `addCall()` only receives the message id—an integer key associated with a method, its class, and signature.

The combined trace of all method calls for all objects quickly reaches Gigabytes in size and cannot be kept in main memory, but writing it to a file would induce a huge runtime overhead. We therefore do not keep the original trace but compute the sequence set for each class online—while tracing. Sequence sets are small (see next Section 3.5 for a discussion of the overhead), kept in memory, and emitted when the program quits.

To compute the sequence set of a class online, each object maintains a window for the last k (incoming or outgoing) calls, which is advanced by code in the prolog of the called method. In addition, a sequence set is associated with every traced class. Whenever a method finds a new sequence—a new window of calls—it adds the sequence to the set of the class. Finally, each class emits its sequence set in XML format.

After the program has quit, we use offline tools to read the sequence sets and analyze them. For our experimental setup, we read them into a relational database.

Computing and emitting sequence sets rather than the original trace has a few disadvantages. To compute sequence sets online, the window size must be fixed for a program run, where sequence sets for many window sizes could be computed offline from a raw trace. While a trace is ordered, a sequence set is not. We therefore lose some of the trace's inherent notion of time.

3.5 Overhead

To validate our claim that capturing call-sequence sets is a lightweight method, we instrumented and traced the programs from the SPEC JVM 98 benchmark suite (SPEC, 1998). We compared the overhead with JCoverage (Morgan, 2004), a tool for coverage analysis that, like ours, works on Java bytecode, and whose results can point to defects.

The SPEC JVM 98 benchmark suite is a collection of Java programs, deployed as 543 class files, with a total size of 1.48 megabytes. Instrumenting them for tracing incoming calls with a window size of 5 on a 3 GHz x86/Linux machine with 1 GB of main memory took 14.2 seconds wall-clock time. This amounts to about 100 kB or 38 class files per second. The instrumented class files increased in size by 26%. Instrumentation thus takes an affordable overhead, even in an interactive setting.

Running an instrumented program takes longer and requires more memory than the original program. Table 1 summarizes the overhead *factors* of the instrumented program relative to the memory consumption and run time of the original program.

The two ray tracers `raytrace` and `mtrt` demonstrate some challenges: tracing them required 380 MB of main memory because they instantiate ten thousands of objects of class `Point`, each of which was traced. This exhausted the main memory, which led to paging and to long run times.

The overheads for memory consumption and runtime varied by two orders of magnitude. At first sight, this may seem prohibitive—even when the overhead was comparable or lower than for JCoverage. We attribute the high overhead in part to the nature of the SPEC JVM 98, which is intended to evaluate Java virtual machines—most programs in the suite are CPU bound and tracing affects them more than, say, I/O-intensive programs.

Table 1. Overhead measured for heap size and time while tracing incoming calls (with window size 5) for the SPEC JVM 98 benchmark. The overhead of our approach (and JCoverage in comparison) is expressed as a factor relative to the original program. The rightmost columns show the number of sequences and the size of their gzip-compressed XML representation

	Memory			Time			Sequences	
Program	original MB	JCoverage factor	**our approach** factor	original seconds	JCoverage factor	**our approach** factor	count	XML KB
check	1.4	1.2	1.1	0.14	10.0	1.5	113	3
compress	30.4	1.2	2.2	5.93	1.7	59.8	85	3
jess	12.1	2.1	17.6	2.17	257.1	98.2	1704	37
raytrace	14.2	1.5	22.7	1.93	380.8	541.6	1489	34
db	20.4	1.4	1.2	11.31	1.5	1.2	127	3
javac	29.8	1.5	1.2	5.46	45.7	31.4	15326	334
mpegaudio	12.8	1.6	1.2	5.96	1.2	27.9	587	13
mtrt	18.4	1.4	18.2	2.06	367.9	574.8	1579	36
jack	13.6	1.7	1.7	2.32	40.5	6.3	1261	28
average		1.5	7.5		122.9	149.2	2477	55
AspectJ	41.8	1.4	1.4	2.37	3.3	3.0	13920	301

The database `db` and the `mpegaudio` decoder benchmarks, for instance, show a small overhead. When we traced the AspectJ compiler for the example in Section 1 (with window size 5), we also observed a modest overhead and consider these more typical for our approach.

4 Relating Call Anomalies to Failures

As described in Section 3.2, a program run yields one sequence set per class. These sequence sets now must be compared across multiple runs—or, more precisely, across passing and failing runs. Our basic claim is that a defective class shows a substantially different sequence set in a passing run than in a failing run. We therefore *rank* classes such that classes whose sequence sets differ the most between passing and failing runs get the highest priority.

For ranking classes we consider *one failing* run $r_{\times}$ and n *passing* runs $r_{\checkmark}^1, \ldots, r_{\checkmark}^n$, where $n \geq 1$. We take into account only one failing run because any additional failing run could be caused by a different defect—something we don't know. We do know, however, that all passing runs are equivalent in the sense that they don't reveal the defect.

Each passing and failing run of a class C is a set of call sequences: $r_{\times}$ is the set of call sequences observed in the failing run, and so on. Since we consider only one class at a time, we don't mention C explicitly and write $r_{\times}$ instead of $r_{\times}(C)$. As an example, we consider five sequences in three passing runs and one failing run:

$$\begin{aligned} r_{\times} &= \{v, w, y, z\} & r_{\checkmark}^2 &= \{x, y, z\} \\ r_{\checkmark}^1 &= \{v, y, z\} & r_{\checkmark}^3 &= \{v, w, z\} \end{aligned}$$

To characterize an individual sequence s in absolute terms, we define the number of passing runs $\#_{\checkmark}$ that contain s, and dual to it, the number of failing runs $\#_{\times}$:

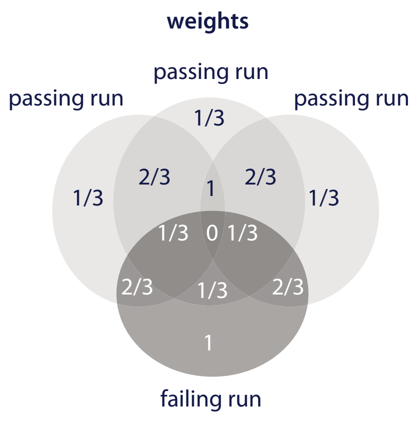
$$\begin{aligned} \#_{\checkmark}(s) &= \Big|\bigcup_{i=1}^{n} \{r_{\checkmark}^i \mid s \in r_{\checkmark}^i\}\Big| \\ \#_{\times}(s) &= |\{r_{\times} \mid s \in r_{\times}\}| \end{aligned}$$

In relative terms, a call sequence s is characterized by the fraction $\#_{\checkmark}(s)/n$ of passing runs where it was observed, and the fraction $\#_{\times}(s)/1$ of failing runs. These two constitute the *weight* $w(s)$ of a sequence:

$$w(s) = \Big|\frac{\#_{\times}(s)}{1} - \frac{\#_{\checkmark}(s)}{n}\Big|$$

The weight of a sequence denotes its responsibility for a fault, expressed as a number in the range 0 to 1. It depends on which sets the sequence is contained in. Figure 5 shows weights for three passing runs and one failing run.

For our example we obtain the counts and weights shown on the right side in Fig. 5. Sequence z is common to all runs and thus has a weight of 0. Sequence w, on the other hand, was observed in the failing run, but only in one out of three passing runs. This earns it a high weight of $^2/_3$. No sequence was observed only in the failing run. The



	v	w	x	y	z
$\#_{\times}(s)$	1	1	0	1	1
$\#_{\checkmark}(s)$	2	1	1	2	3
$w(s)$	$^1/_3$	$^2/_3$	$^1/_3$	$^1/_3$	0

Fig. 5. The weight of a sequence depends on the number of passing and failing runs where it was found. The right side shows the weights for the example of five sequences found in three passing and one failing run

weight of a sequence is high when it is observed in the failing run but in none or few of the passing runs: we then found a "new" sequence in the failing run. Likewise, the weight is high when we find a sequence in many passing runs, but not in the failing run: the sequence is then "missing" in the failing run. A high weight thus is a witness for a different behavior of a class in passing and failing runs. Missing and new sequences are treated dually because a defect could be caused by an extra call, as well as a missing call.

Conversely, the weight of a sequence is low, when it was found in many passing runs as well as in the failing run: we then observed a "common" sequence. It is a witness for similar behavior in passing and failing runs.

Note that the weight of a sequence depends heavily on the number of passing runs. To gain importance, a sequence must be present in many of them. Passing runs thus should be related and show common sequences, but selecting a few unrelated runs does not hurt.

Classes with a similar behavior in passing and failing runs contain mostly light sequences, where the prime suspects are those with many heavy sequences. To identify them, we define the *average sequence weight* for a class:

$$W(C) = \frac{1}{|r|}\sum_{s\in r} w(s) \qquad \text{where } r = r_{\times} \cup r_{\checkmark}^{1} \cup \cdots \cup r_{\checkmark}^{n}$$

In our example, the average sequence weight is $^1/_3$. Because the average sequence weight is independent from the number of sequences observed for a class, we can compare it across classes. The average sequence weight is thus a measure for the importance of a class. When we rank classes by it, classes ranked to the top have a high average weight and are likely to contain a defect. To validate this claim, we conducted two experiments.

5 A Case Study

As described in Section 4, we rank classes based on their average sequence weight and claim that a large weight indicates a defect. To evaluate our rankings, we studied them in an experiment, with the NanoXML parser as our main subject. Our experiments evaluate class rankings along three main axes: incoming versus outgoing calls, various window sizes, and class-based versus object-based traces.

5.1 Object of Study

NanoXML is a non-validating XML parser implemented in Java, for which Do et al. (2004) provide an extensive test suite. NanoXML comes in five development versions[1], each comprising between 16 and 23 classes, and a total number of 33 known faults (Table 2). These faults were discovered during the development process, or seeded by Do and others. Each fault can be activated individually, such that there are 33 variants of NanoXML with a single fault.

Table 2. Characteristics of NanoXML, the subject of our controlled experiment

					Tests		
	Version	Classes	LOC	Faults	All	Failing	Drivers
	1	16	4334	7	214	160	79
	2	19	5806	7	214	57	74
	3	21	7185	10	216	63	76
	5	23	7646	9	216	174	76
total			24971	33		474	

Faults and test cases are related by a fault matrix: for any given fault and test case, the matrix tells whether the test case uncovers the fault. Related test cases share the same driver, which provides general infrastructure for a test.

5.2 Experimental Setup

Our experiment simulates the following situation: for a fixed program, a programmer has one or more passing test cases, and one failing test case. Based on traces of the passing and failing runs, our techniques ranks the classes of the program. The ranking aims to place the faulty class as high as possible.

In our experiment, we know the class that contains the defect (our techniques, of course, do not); therefore, we can assess the ranking. We express the quality of a ranking as the *search length*—the number of classes above the faulty class in the ranking. The best possible ranking places the faulty class at the top (with a search length of zero).

[1] We could not use Version 4 because it lacks known faults for experiments.

To rank classes, we needed at least one passing run for every failing run. However, we wanted to avoid comparing totally unrelated program runs. For each ranking we therefore selected a set of program runs from the suite of programs that met the following conditions:

– We analyze a version of NanoXML with *one known defect*, which is located in a single class.
– As *failing run*, we used a test case that uncovered the known defect.
– As *passing runs*, we selected *all* test cases that did not uncover the known defect.
– All test cases for passing and failing runs must use the *same test driver*. This limits the number of passing runs to those that are semantically related to the failing run.

Altogether, we had 386 such sets (Table 2). The test suite contains 88 more failing runs for which we could not find any passing run. This can happen, for example, when a fault always causes a program to crash such that no passing run can be established.

For each of the failing runs with one or more related passing runs, we traced their classes, computed their sequence sets, and ranked the classes according to their average sequence weight. The rankings were repeated in several configurations:

– Rankings based on class and object traces (recall Section 3.2)
– Rankings based on incoming and outgoing calls (recall Section 3.3).
– Rankings based on 10 window sizes: 1 to 10.

We compared the results of all configurations to find the one that minimizes the search length, and thus provides the best recommendations for defect localization.

5.3 Threats to Validity

Our experiments are not exhaustive—many more variations of the experiment are possible. These variations include other ways to rate sequences, or to trace with class-specific window sizes rather than a universal size. Likewise, we did not evaluate programs with multiple known defects or defects whose fix affects several classes.

The search lengths reported in our results are abstract numbers that don't make potential mistakes obvious. We validated our methods when possible by exploiting known invariants, for example:

– To validate the bytecode instrumentation, we generated Java programs with statically known call graphs and, hence, known sequence sets. We verified that these were indeed produced by our instrumentation.
– When tracing with a window size of one, the resulting sequences for a class are identical for object- and class-based traces: any method called (or initiated) on the object level is recorded in a class-level trace, and vice versa. Hence, the rankings are the same; object- and class-based traces show no difference in search length.

5.4 Discussion of Results

Table 3 summarizes the average search lengths of our rankings for NanoXML, based on different configurations: incoming versus outgoing calls, various window sizes, and

Table 3. Evaluation of class rankings. A number indicates the average number of classes in atop the faulty class in a ranking. The two rightmost columns indicate these numbers for a random ranking when (1) considering only executed classes, (2) all classes

Incoming Calls

	Window Size										Random Guess	
Trace	1	2	3	4	5	6	7	8	9	10	Executed	All
Object	3.66	3.74	4.08	4.07	4.10	4.02	3.91	3.67	3.55	3.49	4.78	9.22
Class	3.66	3.71	3.97	4.05	3.97	4.04	3.97	3.90	3.86	3.85	4.78	9.22

Outgoing Calls

	Window Size										Random Guess	
Trace	1	2	3	4	5	6	7	8	9	10	Executed	All
Object	2.53	2.31	2.19	2.17	2.04	2.00	1.98	2.12	2.15	2.14	4.78	9.22
Class	2.53	2.35	2.22	2.14	2.03	2.04	2.03	2.02	2.22	2.25	4.78	9.22

rankings based on object- and class-based traces. The search length is the number of classes atop of the faulty class in a ranking.

For a ranking to be useful, it must be at least better than a random ranking. Each search length in Table 3 is an average over 386 program runs (or rankings). On average, each run utilizes 19.45 classes from which 10.56 are actually executed (excluding the test driver). Random placing of the faulty class would result in an average search length of $(19.45 - 1)/2 = 9.22$ classes, and 4.78, respectively.

All rankings in our experiment are noticeably better than random rankings. They are better even if a programmer had had the additional knowledge of which classes were never executed.

Comparing sequences of passing and failing runs is effective in locating defects.

Sequences vs. Coverage. Previous work by Jones et al. (2002) has used coverage analysis to rank source code statements: statements more often executed in failing runs than in passing runs rank higher. Since we are ranking classes, the two approaches are not directly comparable.

Ranking classes based on incoming calls with a window size of one is identical to method coverage: the sequence set of a class holds exactly those methods of the class that were called, hence executed. The corresponding search length of 3.66 is the smallest in the table for incoming calls. This suggests that incoming calls perform worse than coverage analysis for defect localization.

The picture is reversed for outgoing calls. Here the search length for a window size of one is the highest in the table. Sequences of calls thus perform better than individual calls.

Incoming calls provide no help for finding defects. Comparing sequences of length 2 or greater always performs better than sequences of length 1 for outging calls.

Classes vs. Objects. Tracing on the object level (rather on the simpler class level) offered no advantage for incoming calls, and only a slight advantage for outgoing calls. We attribute this to the few objects NanoXML instantiates per class and the absence of threads. Both would lead to increased non-deterministic interleaving of calls on the class level, which in turn would lead to artificial differences between runs.

Object-based traces are at least slightly better defect locators than class-based traces. For multi-threaded programs, object-based traces should yield a greater advantage.

Window Size. Incoming calls sequences show no strict relation between window size and search length. There is a trend of an increasing search length when going from a window size of one to 5, and a trend of decreasing search length when moving from 5 to 10.

Outgoing calls show a clear and opposite trend: search lengths are the shortest for window sizes around 7 and increase towards smaller and wider windows. Moving from a window size of one to a window size of 7 reduces the search length by 0.5 classes. This supports our claim that longer outgoing call sequences capture essential control flow of a program. Moving towards wider windows probably does not pay off because increasingly fewer long-living objects actually can fill such windows.

Medium-sized windows, collecting 3 to 8 calls, provide the best predictive power.

Outgoing vs. Incoming Calls. Outgoing calls predict faults better than incoming calls. The search length for rankings based on outgoing calls are smaller than those based on incoming calls. Even the worst result for outgoing calls (2.53 for window size of 1) beats the best result for incoming calls (3.66 for window size of 1). This strongly supports our claim (3): the caller is more likely to be defective than the callee.

The inferiority of incoming calls is not entirely surprising: traces for incoming calls show how an object (or a class) is used. A deviation in the failing run from the passing runs indicates that a class is used differently. But the class is not responsible for its usage—its clients are. Therefore, different usage does not correlate with faults.

This is different for outgoing calls, which show how an object (or a class) is implemented. For any deviation here the class at hand is responsible and thus more likely to contain a fault.

Outgoing calls locate defects much better than incoming calls.

Benefits to the Programmer. Tracing outgoing calls with a window size of 6, the average search length for a ranking was 2.00. On the average, a programmer must thus inspect two classes before finding the faulty class—that is, 18.9% of 10.56 executed classes, or 8.7% of all 23 classes.

Fig. 6 shows a cumulative plot of the search length distribution. Using a window of size 7, the defective class is immediately identified in 39% of all test runs (zero search length). In 47% of all test runs, the programmer needs to examine at most one false positive (search length = 1) before identifying the defect.

Because NanoXML is relatively small, each class comprises a sizeable amount of the total application. As could be seen in the example of AspectJ, large applications

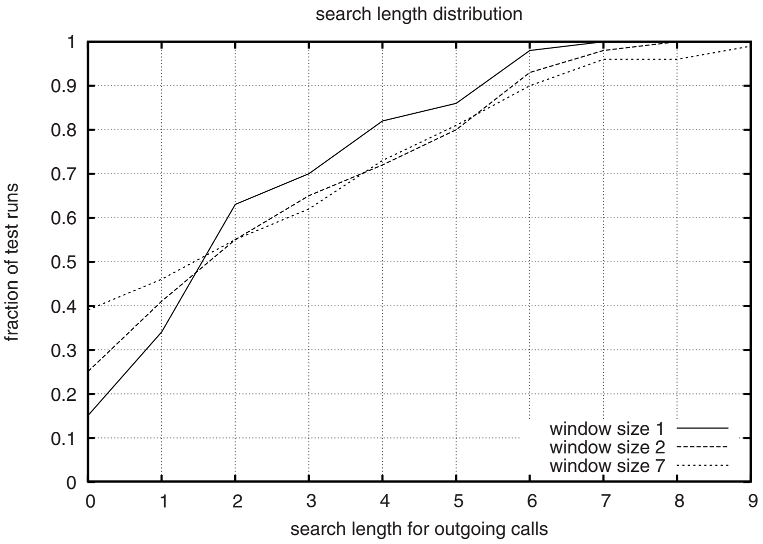


Fig. 6. Distribution of search length for outgoing calls in NanoXML. Using a window size of 7, the defective class is pinpointed (search length 0) in 39% of all test runs

may exhibit vastly better ratios. We also expect larger applications to show a greater separation of concerns, such that the number of classes which contribute to a failure does not grow with the total number of classes. We therefore believe that the results of our controlled experiment are on the conservative side.

In NanoXML, the defective class is immediately identified in 39% of all test runs. On average, a programmer using our technique must inspect 19% of the executed classes (9% of all classes) before finding the defect.

6 Does it Scale?

We have complemented the evaluation of our method with a study of the AspectJ compiler (Kiczales et al., 2001). It differs from NanoXML mainly in its size: AspectJ 1.1.1 consists of 979 classes, representing 112,376 lines of code. Unlike NanoXML, AspectJ does not come pre-packed with a set of defects, and therefore it was not possible to use AspectJ in a systematic evaluation. However, the AspectJ developers have collected bug reports and provide a source code repository that documents how bugs were fixed. From these repositories, we reconstructed passing and failing test cases for our evaluation.

In order to obtain results comparable with our evaluation using NanoXML, we restricted ourself to bugs whose fixes involved only one Java class. Altogether, there are 6 such bugs in the AspectJ bug database, which are shown in Table 4.

For each bug in Table 4, we constructed one passing and one failing run. We traced them for outgoing calls and ranked their classes accordingly.

Table 4. Bugs in AspectJ used for the evaluation. A *Bug ID* refers to the bug description in the AspectJ bug database at `http://bugs.eclipse.org/`

			Size (LOC)	
Bug ID	Version	Defective Class	Class	Fix
29665	1.1b4	org.aspectj.weaver.bcel.BcelShadow	1901	20
29691	1.1b4	org.aspectj.weaver.patterns.ReferencePointcut	294	4
29693	1.1b4	org.aspectj.weaver.bcel.BcelShadow	1901	8
30168	1.1b4	org.aspectj.ajdt.internal.compiler.ast.ThisJoinPointVisitor	225	20
43194	1.1.1	org.aspectj.weaver.patterns.ReferencePointcut	299	4
53981	1.1.1	org.aspectj.ajdt.internal.compiler.ast.Proceed	133	19

Table 5. Evaluation of class rankings for AspectJ. A number indicates the average *search length*: the number of classes atop of the faulty class in a ranking. The two rightmost columns indicate these numbers for a random ranking when (1) considering only executed classes, (2) all classes

	Window Size										Random Guess	
	1	2	3	4	5	6	7	8	9	10	Executed	All
Object	32.4	31.8	30.8	10.2	8.6	23.4	22.6	23.8	24.4	24.0	209	272
Class	32.4	32.2	34.8	12.8	12.4	25.2	24.8	25.2	25.2	25.6	209	272

Our results for window sizes up to 10 are shown in Table 5. The results confirm our previous findings for outgoing calls from the evaluation with NanoXML as subject:

– Rankings based on outgoing calls perform better than random rankings.
– Object-based rankings perform slightly better than class-based rankings.
– Medium-sized windows of 4–7 calls performs best; shorter or wider windows lead to an increased search length. Defect localization benefits from the additional context provided compared to a window size of one.

The difference in search length between a random ranking and a ranking produced by our method is much greater for AspectJ than for NanoXML. Therefore, the benefit for the programmer is even greater: Using a window size of 5, our method on average requires the programmer to examine only 9 of 979 classes (i.e. 0.92% of all classes) until spotting the defect. Again, these results do not necessarily generalize to AspectJ, or to other applications, but they indicate the potential of the approach; they also show that the approach indeed can scale to larger applications.

In our evaluation of AspectJ, we did not consider incoming calls, since they did not prove useful for the NanoXML subject. Also, Table 5 does not take into account the ranking for bug #29665. While bug #29665 is a real bug, the call sequence sets of the defective class were identical in passing and failing runs for all window sizes. A closer inspection of the fix for this bug revealed that the defective method incorrectly returns the same value for the passing and failing run, which is why the defect does not induce a different call sequence. Thus, our method is blind to this defect and cannot localize it.

Defect localization using call-sequence sets scales well to the AspectJ compiler (with excellent results) and is likely to scale to other real-world applications.

7 Related Work

We are by no means the first researchers who compare multiple runs, or analyze function call sequences. The related work can be grouped into the following categories:

Comparing Multiple Runs. The hypothesis that a fault correlates with differences in program traces, relative to the trace of a correct program, was first stated by Reps et al. (1997) and later confirmed by Harrold et al. (1998). The work of Jones et al. (2002) explicitly compares coverage and thus is the work closest to ours. Jones et al. try to locate an error in a program based on the statement coverage produced by several passing and one failing run. A statement is considered more likely to be erroneous the more often it is executed in a failing run rather than in a passing run. In their evaluation, Jones et al. find that in programs with one fault the one faulty statement within a program is almost certainly marked as “likely faulty”, but so is also 5% to 15% of correct code. For programs with multiple faults, this degrades to 5% to 20% with higher variation. Like ours, this approach is lightweight, fully automatic and broadly applicable—but as demonstrated in the evaluation, sequences have a significantly better predictive power.

Intrusion Detection. Our idea of investigating sequences rather than simply coverage was inspired by Forrest et al. (1997) and Hofmeyr et al. (1998)’s work on *intrusion detection.* They traced the system calls of server applications like `sendmail`, `ftpd`, or `lpd` and used the sliding-window approach to abstract them as sequence sets (n-tuples of system calls, where $n = 6, \dots, 10$). In a training phase, they learned the set from normal behavior of the server application; after that, an unrecognized sequence indicated a possible intrusion. As a variation, they also learned sequence that did not match the normal behavior and flagged an intrusion if that sequence was later matched by an application. Intrusion detection is considerably more difficult than defect localization because it has to *predict* anomalous behavior, where we *know* that a program run is anomalous after it failed a test. We found the simplicity of the idea, implementation, and the modest run-time cost appealing. In contrast to their work, though, our approach specifically exploits object orientation and is the first to analyze sequences for defect localization.

Learning Automata. Sekar et al. (2001) note a serious issue in Forrest et al. (1997)’s approach: to keep traces tractable, the window size n must be small. But small windows fail to capture relations between calls in a sequence that are n or more calls apart. To overcome this, the authors propose to *learn finite-state automata* from system call sequences instead and provide an algorithm. The interesting part is that Sekar et al. learn automata from traces where they annotate each call with the caller; thus calls by two different callers now become distinguishable. Using these more context-rich traces, their automata produced about 10 times fewer false positives than the n-gram approach. Learning automata from object-specific sequences is an interesting idea for future work.

Learning APIs. While we are trying to locate defects relative to a failing run, Ammons et al. (2002) try to locate defects relative to *API invariants* learned from correct runs: they observe how an API is used by its clients and learn a finite-state automaton that describes the client's behavior. If in the future a client violates this behavior, it is flagged with an error. A client is only required during the learning phase and the learned invariants can later be used to validate clients that did not even exist during the learning phase. However, as Ammons et al. point out, learning API invariants requires a lot of effort—in particular because context-sensitive information such as resource handles have to be identified and matched manually. With object-specific sequences, as in our approach, such a context comes naturally and should yield better automata with less effort.

Data Anomalies. Rather than focusing on diverging control flow, one may also focus on differing data. *Dynamic invariants,* pioneered by Ernst et al. (2001), is a predicate for a variable's value that has held for all program runs during a training phase. If the predicate is later violated by a value in another program run this may signal an error. Learning dynamic invariants takes a huge machine-learning apparatus and is far from lightweight both in time and space. While Pytlik et al. (2003) have not been able to detect failure-related anomalies using dynamic invariants, a related lightweight technique by Hangal and Lam (2002) found defects in four Java applications. In general, techniques that detect anomalies in data can complement techniques that detect anomalies in control flow and vice versa.

Statistical Sampling. In order to make defect localization affordable for production code in the field, Liblit et al. (2003) suggest statistical sampling: Rather than collecting all data of all runs, they focus on exceptional behavior—as indicated by exceptions being raised or unusual values being returned—but only for a *sampled set.* If such events frequently occur together with failures (i.e. for a large set of users and runs), one eventually obtains a set of anomalies that statistically correlate with the failure. Our approach requires just two instrumented runs to localize defects, but can be easily extended to collect samples in the field.

Isolating Failure Causes. To localize defects, one of the most effective approaches is isolating *cause transitions,* as described by Cleve and Zeller (2005). Again, the basic idea is to compare passing and failing runs, but in addition, the delta debugging technique generates and tests *additional runs* to isolate failure-causing variables in the program state (Zeller, 2002). A cause transition occurs at a statement where one variable ceases to be a cause, and another one begins; these are places where cause-effect chains to the failure originate (and thus likely defects). Due to the systematic generation of additional runs, this technique is precise, but also demanding—in particular, one needs an automated test and a means to extract and compare program states. In contrast, collecting call sequences is far easier to apply and deploy.

8 Conclusion and Consequences

Sequences of method calls locate defective classes with a high probability. Our evaluation also revealed that per-object sequences are better predictors of defects than per-class or global sequences, and that the caller is significantly more likely to be defective than the callee. In contrast to previous approaches detecting anomalies in API usage, our technique exploits object orientation, as it collects method call sequences per object; therefore, the approach is fully generic and need not be adapted to a specific API. These are the results of this paper.

On the practical side, the approach is easily applicable to arbitrary Java programs, as it is based on byte code instrumentation, and as the overhead of collecting sequences is comparable to measuring coverage. No additional infrastructure such as automated tests or debugging information is required; the approach can thus be used for software in the field as well as third-party software.

Besides general issues such as performance or ease of use, our future work will concentrate on the following topics:

Further Evaluation. The number of Java programs that can be used for controlled experiments (i.e. with known defects, automated tests that reveal these defects, and changes that fix the defects) is still too limited. As more such programs become available (Do et al., 2004), we want to gather further experience.

Fine-Grained Anomalies. Right now, we are identifying *classes* as being defect-prone. Since our approach is based on comparing *methods,* though, we could relate differing sequences to sets of methods and thus further increase precision. Another interesting option is to identify anomalies in sequences of basic blocks rather than method calls, thus focusing on individual statements.

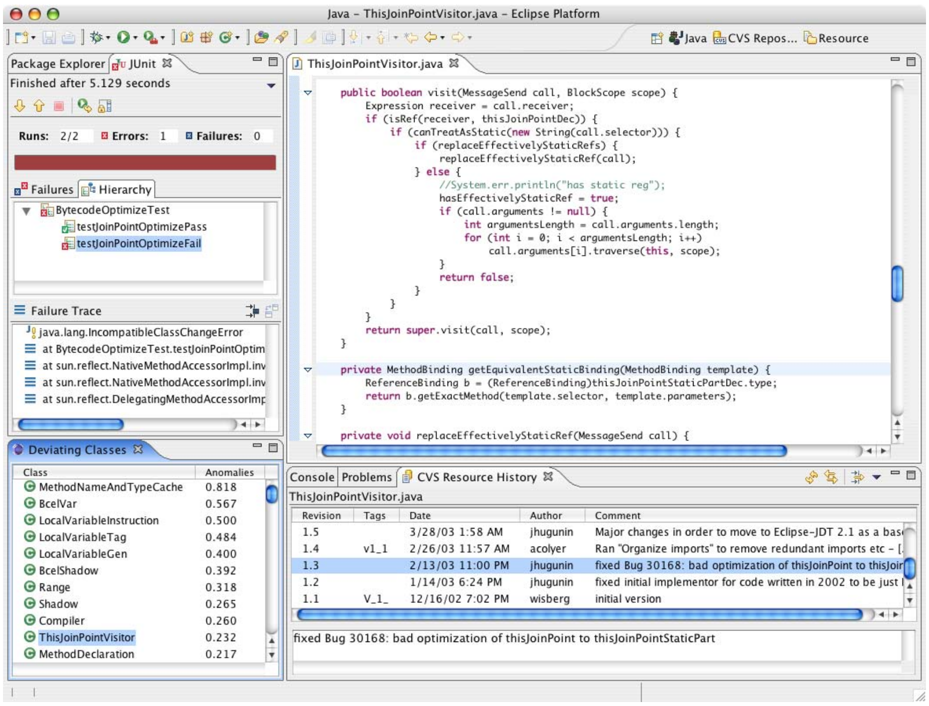
Sampled Calls. Rather than collecting every single method call, our approach could easily be adapted to *sample* only a subset of calls—for instance, only the method calls of a specific class, or only every 100th sequence (Liblit et al., 2003). This would allow to use the technique in production code and thus collect failure-related sequences in the field.

Exploiting Object Orientation. Our approach is among the first that explicitly exploits object orientation for collecting sequences. Being object-aware might also be beneficial to related fields such as intrusion detection or mining specifications.

Integration with Experimental Techniques. Anomalies in method calls translate into specific objects and specific moments in time that are more interesting than others. These objects and moments in time could be good initial candidates for identifying failure-inducing program state (Zeller, 2002).

An Eclipse Plugin. Last but not least, we are currently turning our prototype into an Eclipse plugin called AMPLE (for "Analyzing Method Patterns to Locate Errors"). As soon as a JUnit test fails, AMPLE displays a list showing the most likely defective classes at the top—as in the AspectJ example (Fig. 7). We plan to make AMPLE publicly available in the second half of this year. For future and related work regarding defect localization, see

`http://www.st.cs.uni-sb.de/dd/`



```
Finished after 5.129 seconds
Runs: 2/2   Errors: 1   Failures: 0

BytecodeOptimizeTest
    testJoinPointOptimizePass
    testJoinPointOptimizeFail

java.lang.IncompatibleClassChangeError
  at BytecodeOptimizeTest.testJoinPointOptim
  at sun.reflect.NativeMethodAccessorImpl.inv
  at sun.reflect.NativeMethodAccessorImpl.inv
  at sun.reflect.DelegatingMethodAccessorImp
```

Class	Anomalies
MethodNameAndTypeCache	0.818
BcelVar	0.567
LocalVariableInstruction	0.500
LocalVariableTag	0.484
LocalVariableGen	0.400
BcelShadow	0.392
Range	0.318
Shadow	0.265
Compiler	0.260
ThisJoinPointVisitor	0.232
MethodDeclaration	0.217

```java
    public boolean visit(MessageSend call, BlockScope scope) {
        Expression receiver = call.receiver;
        if (isRef(receiver, thisJoinPointDec)) {
            if (canTreatAsStatic(new String(call.selector))) {
                if (replaceEffectivelyStaticRefs) {
                    replaceEffectivelyStaticRef(call);
                } else {
                    //System.err.println("has static reg");
                    hasEffectivelyStaticRef = true;
                    if (call.arguments != null) {
                        int argumentsLength = call.arguments.length;
                        for (int i = 0; i < argumentsLength; i++)
                            call.arguments[i].traverse(this, scope);
                    }
                    return false;
                }
            }
        }
        return super.visit(call, scope);
    }

    private MethodBinding getEquivalentStaticBinding(MethodBinding template) {
        ReferenceBinding b = (ReferenceBinding)thisJoinPointStaticPartDec.type;
        return b.getExactMethod(template.selector, template.parameters);
    }

    private void replaceEffectivelyStaticRef(MessageSend call) {
```

ThisJoinPointVisitor.java

Revision	Tags	Date	Author	Comment
1.5		3/28/03 1:58 AM	jhugunin	Major changes in order to move to Eclipse-JDT 2.1 as a bas
1.4	v1_1	2/26/03 11:57 AM	acolyer	Ran "Organize imports" to remove redundant imports etc – [
1.3		2/13/03 11:00 PM	jhugunin	fixed Bug 30168: bad optimization of thisJoinPoint to thisJoir
1.2		1/14/03 6:24 PM	jhugunin	fixed initial implementor for code written in 2002 to be just I
1.1	V_1_	12/16/02 7:02 PM	wisberg	initial version

fixed Bug 30168: bad optimization of thisJoinPoint to thisJoinPointStaticPart

Fig. 7. For the AspectJ bug of Section 2, Eclipse ranks likely defective classes (bottom left)

Acknowledgments. Gregg Rothermel made the NanoXML test suite accessible. Tom Zimmermann provided precious insights into the AspectJ history. Holger Cleve, Stephan Neuhaus, and the anonymous reviewers provided valuable comments on earlier revisions of this paper.

Bibliography

Glenn Ammons, Rastislav Bodík, and Jim Larus. Mining specifications. In *Conference Record of POPL'02: The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, Portland, Oregon, January 16–18, 2002.

Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proc. 27th International Conference of Software Engineering (ICSE 2005)*, St. Louis, USA, 2005. to appear.

Markus Dahm. Byte code engineering with the JavaClass API. Technical Report B-17-98, Freie Universität Berlin, Institut für Informatik, Berlin, Germany, July 07 1999. URL `http://www.inf.fu-berlin.de/~dahm/JavaClass/ftp/report.ps.gz`.

Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Infrastructure support for controlled experimentation with software testing and regression testing techniques. In *International Symposium on Empirical Software Engineering*, pages 60–70, Redondo Beach, California, August 2004.

Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001. A previous version appeared in *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

Stephanie Forrest, Steven A. Hofmeyr, and Anil Somayaji. Computer immunology. *Communications of the ACM*, 40(10):88–96, October 1997. ISSN 0001-0782.

Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 291–301, New York, May 19–25 2002. ACM Press.

Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. An empirical investigation of program spectra. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98), ACM SIGPLAN Notices*, pages 83–90, Montreal, Canada, July 1998. Published as ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98), ACM SIGPLAN Notices, volume 33, number 7.

Steven A. Hofmeyr, Stephanie Forrest, and Somayaji Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proc. International Conference on Software Engineering (ICSE)*, pages 467–477, Orlando, Florida, May 2002.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Jorgen Lindskov Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, 2001.

Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Proc. of the SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, San Diego, California, June 2003.

Peter Morgan. JCoverage 1.0.5 GPL, 2004. URL `http://www.jcoverage.com/`.

Brock Pytlik, Manos Renieris, Shriram Krishnamurthi, and Steven Reiss. Automated fault localization using potential invariants. In Michiel Ronsse, editor, *Proc. Fifth Int. Workshop on Automated and Algorithmic Debugging (AADEBUG)*, Ghent, Belgium, September 2003. URL `http://xxx.lanl.gov/html/cs.SE/0309027`.

Steven P. Reiss and Manos Renieris. Encoding program executions. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE-01)*, pages 221–232, Los Alamitos, California, May12–19 2001. IEEE Computer Society.

Thomas Reps, Thomas Ball, Manuvir Das, and Jim Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 432–449. Lecture Notes in Computer Science Nr. 1013, Springer–Verlag, September 1997.

R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In Francis M. Titsworth, editor, *Proceedings of the 2001 IEEE Symposium on Security and Privacy (S&P-01)*, pages 144–155, Los Alamitos, CA, May 14–16 2001. IEEE Computer Society.

SPEC. SPEC JVM 98 benchmark suite. Standard Performance Evaluation Corporation, 1998.

Andreas Zeller. Isolating cause-effect chains from computer programs. In William G. Griswold, editor, *Proceedings of the Tenth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-02)*, volume 27, 6 of *Software Engineering Notes*, pages 1–10, New York, November 18–22 2002. ACM Press.

Extending JML for Modular Specification and Verification of Multi-threaded Programs

Edwin Rodríguez[1], Matthew Dwyer[2], Cormac Flanagan[3], John Hatcliff[1], Gary T. Leavens[4], and Robby[1]

[1] Department of Computing and Information Sciences, Kansas State University
{edwin, hatcliff, robby}@cis.ksu.edu
[2] Department of Computer Science and Engineering, University of Nebraska-Lincoln
dwyer@cse.unl.edu
[3] Computer Science Department, University of California at Santa Cruz
cormac@cs.ucsc.edu
[4] Department of Computer Science, Iowa State University
leavens@cs.iastate.edu

Abstract. The Java Modeling Language (JML) is a formal specification language for Java that allows developers to specify rich software contracts for interfaces and classes, using pre- and postconditions and invariants. Although JML has been widely studied and has robust tool support based on a variety of automated verification technologies, it shares a problem with many similar object-oriented specification languages—it currently only deals with sequential programs. In this paper, we extend JML to allow for effective specification of multi-threaded Java programs. The new constructs rely on the non-interference notion of *method atomicity*, and allow developers to specify locking and other non-interference properties of methods. Atomicity enables effective specification of method pre- and postconditions and supports Hoare-style modular reasoning about methods. Thus the new constructs mesh well with JML's existing features. We validate the specification language design by specifying the behavior of a number of complex Java classes designed for use in multi-threaded programs. We also demonstrate that it is amenable to automated verification using model checking technology.

1 Introduction

The use of rich source-level specification languages for expressing correctness properties of object-oriented programs is growing in practice. Specification languages such as the Java Modeling Language (JML) [1, 2, 3, 4] and Spec# [5] provide a wide range of light-weight annotations (e.g., specifying non-nullness of variables of reference type) as well as constructs for writing specifications of full functional behaviors of class implementations that can be checked by a variety of verification technologies including static analysis, run-time monitoring, model checking, and theorem-proving. JML is a behavioral interface specification language that allows developers to specify both the syntactic and behavioral interface of a portion of Java code. It supports the design by contract paradigm [6] by including notation for pre- and postconditions and invariants. JML uses Java's expression syntax and adds features for: universal (`\forall`) and existential (`\exists`) quantification over object instances as well as basic types, such as integers, and constructs for expressing properties of heap allocated data (such as `\reach` which returns the set of objects reachable from a particular reference).

JML has proved to be an effective vehicle for bringing together a number of research teams [1] seeking to (a) extend the logical foundations of specification formalisms needed for addressing semantically complex language features such as dynamic dispatch, exceptions, dynamic object creation, and (b) build tool support for automated and computer-assisted reasoning about real-world Java applications. However, despite the success of JML in specifying programs written in sequential Java and its Java Card dialect, JML's support for concurrency "is still in its infancy" [4].

Although many interesting programs are sequential, the flexibility that accompanies concurrent programming in terms of further modularizing the design (thread modularity), means that most moderately complex systems are programmed with some sort of concurrency modality (multi-threading, multi-processing, etc.). Moreover, multi-threading capabilities are becoming more accessible to programmers since languages like Java and C# provide direct language support for threads, while other languages provide sophisticated support via libraries (e.g., POSIX threads).

Most existing specification and checking tools for multi-threaded programs focus on properties such as absence of race conditions, establishing mutual exclusion, and simple event ordering and temporal properties (e.g., capturing proper ordering of calls to APIs). However, they typically ignore strong functional properties and complex data structure invariants. These inadequacies stem from the challenges of dealing with thread interference in the manipulation of shared (heap) data. One cannot simply apply Hoare-style logics using method pre- and postconditions to reason modularly, because method execution is often *non-serial*—the actions of other threads may interfere with the thread executing the method and thus render invalid assumptions captured in method preconditions and guarantees captured in postconditions.

Due to the pervasiveness of multi-threading and the increasing use of multi-threaded object-oriented code in embedded and mission- and safety-critical applications, it is necessary to extend sequential specification languages like JML to support specification and reasoning about multi-threaded programs. Furthermore, these extensions should allow both light-weight annotations and more complete, functional specifications, and should work with multiple different reasoning tools. This paper advances toward these goals by making the following contributions:

– We identify situations in which the current JML fails to enable effective specification and modular reasoning for multi-threaded programs.
– We identify specification forms that we and other researchers have found useful for multi-threaded programs. This includes (a) various light-weight

annotations that can be leveraged by automated checking technologies and (b) the use of atomicity specifications to achieve modular reasoning about methods.
– We show how to integrate these forms into JML in a way that enables both reasoning about data values and concurrency concerns.
– We validate the design of this enhanced version of JML by using it to specify properties of a number of Java libraries designed for concurrency, including most of the concurrent data structure classes from `java.util.concurrent`, which includes some very intricate concurrent Java code (the full collection of these specified examples are posted on our project web-site [7]).
– We establish that these specification formalisms are amenable to effective automated verification, by providing experimental results of checking these using a verification framework built on top of our Bogor software model checker (extended from our previous work on model checking JML [8] and atomicity specifications [9]).

Our approach does not explicitly deal with Java's relaxed memory model [10], and instead assumes a sequential consistent memory model. This assumption is sound for programs that are free of race conditions, and race-freedom can be verified via separate analyses [11].

Although we have used JML for this work, we believe the ideas could also be adapted to other specification languages such as Spec# and Eiffel. However, we leave detailed investigation of such adaptation for future work.

In the next section, we describe the problems addressed in this work, in particular, the limitations of JML for concurrent programs. Section 3 gives background on the concept of atomicity, on which our approach is based. Section 4 introduces the new set of annotations, giving examples and an assessment of the issues addressed by each annotation. Section 5 reports on an evaluation of using JML extensions to specify Java classes and of checking such specifications using a customized model checker. Section 6 surveys related work, and Sec. 7 concludes.

2 The Problem

In this section we discuss the semantical and expressiveness problems that need to be solved to allow effective specification and reasoning about both data values and proper thread behavior in a multi-threaded program.

2.1 Interference

Interference causes problems that affect modular reasoning about data values in multi-threaded programs. These semantical problems are best illustrated by an example.

Consider the method in Fig. 1. This is a method from a concurrent linked queue class, and is adapted from Lea's book [12]. This method extracts an element from the queue. The figure shows an invariant for the class at the beginning

```
public class LinkedQueue {
    protected /*@ spec_public non_null @*/ LinkedNode head;
    protected /*@ spec_public non_null @*/ LinkedNode last;
    //@ public invariant head.value == null;

    /*@ public normal_behavior
      @   requires head == last;
      @   assignable \nothing;
      @   ensures \result == null;
      @ also public normal_behavior
      @   requires head != last;
      @   assignable head, head.next.value;
      @   ensures head == \old(head.next) && \result == \old(head.next.value);
      @*/
    public synchronized Object extract() {
        synchronized (head) {
            Object x = null;
            LinkedNode first = head.next;
            if (first != null) {
                x = first.value;
                first.value = null;
                head = first;
            }
            return x;
        }
    }
```

Fig. 1. JML Specification for the method `extract()`

of the class declaration. Invariants must be satisfied by the instances of the class at every method's pre- and post-state. The figure also shows a behavioral specification of the method written in JML, without using any of the extensions proposed in this paper. JML's annotations are written as special Java comments that begin with an at-sign (`@`). The specification of the `extract()` method appears just before its header. This specification is comprised of two `normal_behavior` specification cases, each of which has a `requires` clause, which gives its precondition, an `assignable` clause, which gives a frame axiom, and an `ensures` clause, which gives its postcondition. The first case applies when the list is empty at the beginning of the method's execution (`head == last`), and the other when the list is non-empty (`head != last`). In the first case the method must return `null`, without assigning to any locations. Otherwise, the method updates `head` to the next node in the queue (`head.next`), and must return the object that was contained in that node (since `head` is a sentinal, as described by the invariant). To satisfy the invariant the method must also make the value of the new head `null`. The method may assign to both `head` and `head.next.value` to achieve this behavior.

The meaning of a JML method specification with two or more specification cases, combined with "`also`", is that the caller has to satisfy the disjunction of the preconditions in the given specification cases, and the implementation has to satisfy the postconditions of all specification cases for which the preconditions held [4, 13]. Thus, in Fig. 1 the caller has no obligations, since the disjunction of the preconditions of the two specification cases is the tautology "`head == last || head != last`".

Internal Interference. Although the specification given in Fig. 1 seems sensible, it is wrong for a multi-threaded environment, because it does not account for interference. An example of interference is depicted in Fig. 2. The queue is empty in the method's pre-state. Since this method is `synchronized`, the first instruction it executes is to acquire the lock on `this`. Then, in this trace, another thread is immediately scheduled that executes a call to method `insert(Object)`, which is synchronized on a lock other than `this` (to allow a fine grained degree of concurrency), and executes it to completion. When the `extract()` call in the first thread resumes the queue is no longer empty and the method will find a non-null element to extract. Since the list was empty in the pre-state, the first specification case should apply, but the interference causes a non-null value to be returned which violates the postcondition of that case.

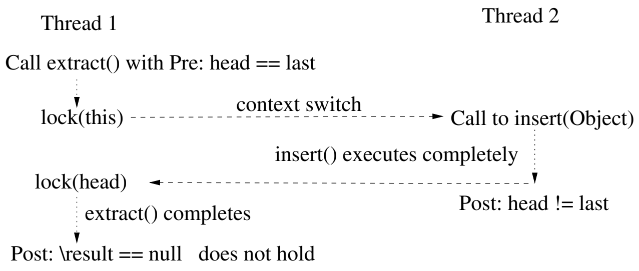


Fig. 2. Execution of `extract()` call interleaved with a call to `insert(Object)`

We call the problem illustrated by Fig. 2 *internal interference*. This problem arises when another thread affects the current thread's execution of a method, by changing data that the method can observe. Standard Hoare logic does not allow for such interference, and thus when reasoning about the correctness of the implementation of a method in Hoare logic, one assumes that properties, such as the method's precondition, do not change except by actions of the method itself.

Runtime and static analysis tools for sequential programs exploit this semantics when they work with just two states: the *pre-state* (at the beginning of the method's execution) and the *post-state* (at the end). In a multi-threaded setting, however, such analyses must consider all possible interleavings to safely account for possible interference. This is considerably more expensive than restricting reasoning to pre- and post-states; furthermore, it is non-modular.

External Interference. *External interference* happens when another thread makes observable state changes between a method call and the method's entry, or between the method's exit and the caller's resumption. Just as internal interference can invalidate standard Hoare-style reasoning about the correctness of a method implementation, external interference can disrupt reasoning about the correctness of client code that calls that method.

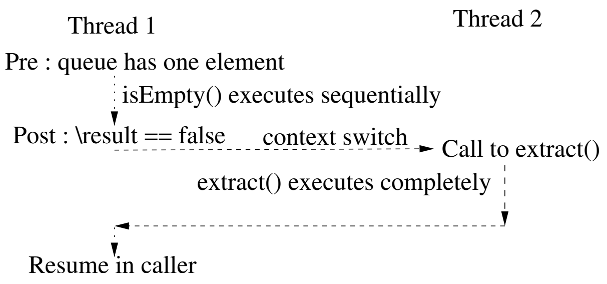

Thread 1
Thread 2
Pre : queue has one element
isEmpty() executes sequentially
Post : \result == false
context switch
Call to extract()
extract() executes completely
Resume in caller

Fig. 3. Execution of `extract()` call interleaved immediately after call to `isEmpty()`

Figure 3 illustrates the problem. Suppose the queue has exactly one element. The `isEmpty()` method executes without interleaving and returns false. However, between the return of `isEmpty()` and the resumption of the caller, another thread interleaves a call to `extract()`, which removes the lone element. The result is that upon resumption of the caller's thread, the postcondition no longer describes the queue correctly, as the queue is now empty. Similar interference can happen with respect to the precondition established by the caller and observed upon method entry. As can be seen from this example, external interference breaks the modularity of reasoning, thereby rendering existing sequential analysis techniques inapplicable.

More generally, these types of interference mean that specifications cannot serve as behavioral abstractions in the presence of concurrency. That is, one cannot use method specifications to reason about the correctness of an implementation without knowing some details of its calling context, or reason about calls to the method without knowing some details of its implementation. To fix these problems, while still allowing modular reasoning, the specification language must be enriched.

2.2 Expressing Thread-Safe Behavior

There are a variety of ways that one might incorporate features into a behavioral specification language to support specification in the presence of concurrency. Our approach has been motivated primarily by the results of a survey of existing Java implementations to understand the mechanisms used by developers to assure proper concurrent execution, and of existing specification features in the JML language. We wanted to minimally extend JML, and yet enable modular behavioral specification of a wide range of existing multi-threaded Java implementations.

Our fundamental observation is that while programmers may use a variety of mechanisms to achieve thread safety, the core notion of safety is one of non-interference. Interference can be avoided through the use of synchronization, which prevents unwanted interleaving, or by controlling access to data, which prevents unwanted access to otherwise shared objects.[1] When we speak of *thread safety*, we mean either synchronization or controlled access to data, or some combination of both. We have identified several fundamental notions that must be included in JML or similar languages to support thread-safe behavioral specification.

Locking Specifications. Programmers use a variety of locking disciplines and the language must be rich enough to capture that variety. It is necessary to allow specification of:

– what locks a method will acquire and release during its execution,
– what locks protect particular parts of an object's state,
– that some objects are used as locks, and when such lock objects are locked,
– the set of locks held by the current thread, and
– that an object is protected by some lock held by the current thread.

We have also found it necessary to specify the conditions under which a method may block [14].

Data Confinement Specifications. Excessive locking can reduce parallelism and hence performance. For this reason, many implementations avoid locking and instead rely on properties of a program's data layout to ensure thread safety. It is necessary to allow specification of:

– what aliasing and ownership patterns exist among objects [15, 16, 17],
– that an object is local to a thread, and
– the effect of a method's execution on existing locations (*i.e.*, a frame axiom [18]).

Serializability Specifications. Locking and data confinement specifications are useful in specifying the conditions under which multi-threaded executions are equivalent to sequential executions. It is necessary to allow specification of this high-level *serializability* property of methods. We have found two different strengths of such specification to be useful in practice: atomicity and independence. These concepts and the extensions to JML that support them are described in the next sections.

3 Background on Atomicity and Independence

Our approach to addressing the interference problems above is based on the concepts of atomicity [19, 20] and independence [21]. A region of code statements (*e.g.*, a method body) is said to be *atomic* if the statements in the region are *serializable*—that is, if for any execution trace containing the region's statements

[1] We do not treat extra-program forms of concurrency control, such as scheduling.

(possibly interleaved with statements executed by other threads) there is an equivalent execution trace where the region's statements are executed sequentially (*i.e.*, executed without any interleavings from other threads). If a code region is atomic, then it is sound to reason about its actions as if they occur in a single atomic step—in essence, allowing one to use traditional sequential reasoning techniques on the code region. From another point of view, instead of having to consider a number of intermediate states produced by thread interleavings, for an atomic region it is sound to consider only two states: the *pre-state* before the conceptual single atomic step begins, and the *post-state* after the conceptual single atomic step completes. That is, any interference from the other threads is benign, however, the single atomic step may interfere with other threads' computations.

There are many ways to establish the atomicity of a code region. In the next two subsections we describe two popular approaches; these approaches will motivate the JML notations that we develop in the following sections.

3.1 Lipton's Reduction Theory

Lipton introduced the theory of left/right movers to aid in proving properties about concurrent programs [19]. In Lipton's model, a code region is thought of as a sequence of primitive statements (*e.g.*, Java bytecodes), which he called *transitions*. Proofs about the transitions in a program can be made simpler if one is allowed to assume that a particular sequence of transitions is indivisible. To conclude that a program, P, which contains a sequence of transitions, S, is equivalent to the reduced program, P/S, in which S is modeled as one indivisible transition, Lipton proposed the notion of a commuting transition. A *commuting transition* is a transition that is either a right mover or a left mover. Intuitively, a transition, α, is a *right (left) mover* if, whenever α is followed (preceded) by another transition, β, of a different thread, then α and β can be swapped without changing the resulting state. Concretely, a lock acquire, such as the beginning of a Java synchronized block, is a right mover, and the lock release at the end of such a block is a left mover. Any read or write to a variable or field that is properly protected by a lock is both a left and right mover, which is termed a *both mover*.

To illustrate the application of these ideas, we repeat the example given in [20]. Consider a method m that acquires a lock, reads a variable `x` protected by that lock, updates `x`, and then releases the lock. Suppose that the transitions of this method are interleaved with transitions E_1, E_2, and E_3 of other threads, as shown at the top of Fig. 4. Because the actions of the method m are movers (`acq` and `rel` are right and left movers, respectively, and the lock-protected assignment to `x` is a both mover), Fig. 4 implies that there exists an equivalent execution (shown at the bottom of the figure), where the operations of m are not interleaved with operations of other threads. Thus, it is safe to reason about the method as executing in a single atomic step.

One can define an *atomic region* as one that satisfies the pattern of statements $\mathsf{R}^*\mathsf{N}^?\mathsf{L}^*$, where R^* denotes 0 or more right mover statements, L^* denotes 0 or more

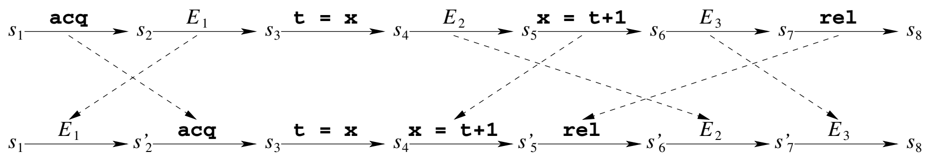

s_1 acq s_2 E_1 s_3 t = x s_4 E_2 s_5 x = t+1 s_6 E_3 s_7 rel s_8
s_1 E_1 s'_2 acq s_3 t = x s_4 x = t+1 s'_5 rel s'_6 E_2 s'_7 E_3 s_8

Fig. 4. Left/Right movers and atomic blocks

left mover statements, and $\mathsf{N}^?$ denotes 0 or 1 statements that are neither left nor right movers. That is, an atomic region can contain at most one non-commuting (*i.e.*, possibly interfering) statement. The block shown in Fig. 4 matches this pattern in the following way: the statement `acq` is right mover (R), the statement `t = x` is both right and left mover so it can stand for right (R), the statement `x = t+1` is both right and left mover so it can stand for left (L), and the statement `rel` is left mover. Thus we get RRLL, which fits the pattern.

In other words, an atomic region can have a single externally-observable effect in its body while it is executed. However, note that an atomic method can have multiple accesses to heap objects as long as they are either thread local or lock-protected accesses [22]. This is because accesses to objects local to a thread and to objects protected by locks cannot be observed by other threads until these objects become shared or until the locks are released, respectively. Similarly, lock-acquires and lock-releases on an object that is already locked by a thread cannot be observed by other threads until that lock is released.

Lipton also stated two technical conditions necessary to prove that the set of final states of a program P equals the set of final states of the reduced program P/S. The first of these, R1, [19–p. 719] states that "if S is ever entered then it should be possible to eventually exit S." This is a fairly strong liveness requirement that can be violated if, for example, S contributes to a deadlock or livelock, or fails to complete because it performs a Java `wait` and is never notified. Restriction R2 is that "the effect of statements in S when together and separated must be the same." This is essentially stating an interference-free property from the other threads for S: any interleavings between the statements of S do not affect the final store it produces.

3.2 Independent Statements

Statements that are both movers are also referred to as *independent statements* since they can commute both to the left and the right of other program statements. We say that a code region is *independent* if each statement within the region is independent. Thus, an independent code region satisfies the pattern of statements I^*, where I^* denotes 0 or more both mover statements. An independent region is totally non-interfering, and thus is trivially atomic.

Our specification methodology for reasoning about method calls within contexts will rely on the fact that independent regions have pleasing composability properties. Sequentially composing two independent regions yields an independent region, but composing two atomic regions may not yield an atomic region if each region contains a non-mover. Moreover, the sequential composition of an independent region and an atomic region is an atomic region (since both movers can serve as either left or right movers). Intuitively, calling a method M_2 from inside of a method M_1 represents the sequential composition of three code regions (the part of M_1 before the call, the body of M_2, the rest of M_1). Thus, if one takes an atomic method, M_1, and inserts into its body a call to an independent method, M_2, then M_1 remains atomic. However, if the inserted call is to an atomic method, M_3, then M_1 does not necessarily remain atomic.

Finally, we note that for a region to be independent, one does not have to establish Lipton's R1 liveness condition to ensure the existence of an equivalent serialized trace (it is the asymmetric nature of the left and right movers in the criterion for atomic regions that necessitates the liveness condition, R1).

4 Introducing Concurrency into JML

Our approach for introducing concurrency into JML is to separate the concern of property specification for methods into two parts: (1) atomicity and independence properties, and (2) specification of sequential (or functional) behavior. This is an old idea, but quite useful. What we claim is new is the language design, which provides necessary and sufficient constructs to specify a wide range of multi-threaded programs.

In what follows we describe just the new JML constructs relating to specification of atomicity and independence and to expressing locking and other properties specific to multi-threaded programs. We illustrate the new constructs using examples from Doug Lea's and Java 1.5's concurrent libraries.

4.1 Locking Notations

Locking is an important aspect of concurrent behaviors, since it is the usual mechanism used to achieve atomicity. So we need several notations that allow for the specification of locking behaviors. These notations also allow modular checking of atomicity specifications (described in Sec. 4.3).

JML already has several notations that can be used to specify information about locking. The `monitors_for` clause allows specifying the locks that protect the access to a given field. The syntax of this clause is:

⟨monitors-for-clause⟩ ::= `monitors_for` ⟨ident⟩ `<-` ⟨store-ref-list⟩ `;`

The meaning is that all of the (non-null) locks named in the ⟨store-ref-list⟩ must be held by a thread in order to access the field `ident`. (In JML, a ⟨store-ref-list⟩ is a comma-separated list of access expressions, which includes identifiers, field and array accesses, and various patterns [4].) An example in Sec. 4.4 demonstrates the use of the `monitors_for` clause.

Finally, the `\lockset()` expression returns an object, of type `JMLObjectSet`, that represents the set of all locks held by the current thread.

The first new construct we add to JML is the `locks` clause. This clause can appear in the body of a specification case after the heading part (in which the `requires` clause appears). Its syntax specifies a list of locks:

⟨locks-clause⟩ ::= `locks` ⟨store-ref-list⟩`;`

Figure 5 shows the use of the `locks` clause in the method `extract()`. The `locks` clause accomplishes two different purposes. First, it is an explicit statement of the locks that the current method acquires (and releases) during its execution. The meaning is that, on any given execution (where the precondition is satisfied), the method will lock all the locks in the given list. Second, the `locks` clause states an implicit condition for independence. In general, a `locks` clause of the form:

`locks` $l_1, \ldots, l_n$`;`

desugars to an ensures clause of the form:

`ensures \old(\lockset().has(`l_1`) && ... && \lockset().has(`l_n`))`
`        ==> \independent;`

Therefore, if another method calls the method with the `locks` clause in a context where all the locks in the list are held, then the callee must be guaranteed to be independent (see Sec. 4.4). In this sense, the `locks` clause gives a lower bound on the set of locks that must be held by callers to ensure independent execution when the method is called. This can help verify a caller's atomicity specification. Conversely, the `locks` clause limits the locks that an implementation of the method may try to acquire (when the precondition of the specification case it appears in holds); thus for the implementation the `locks` clause gives an upper bound on the set of locks that the method may try to acquire.

For an instance (static) method, the `locks` clause has a default value of `this` (the class object) if the method is specified as `synchronized`, otherwise it defaults to `\nothing`. The default for synchronized methods is useful because many concurrent methods in Java synchronize on `this`.

Finally, we add a predicate, `\lock_protected`, with the following syntax.

⟨lock-protected-expression⟩ ::= `\lock_protected(`⟨store-ref⟩`)`

An expression such as `\lock_protected(`o`)` states that the object referenced by o is access-protected by some nonempty set of locks, and all of those locks are held by the current thread. Notice that this is a very strong property: the access is restricted with respect to the object, not the reference variable, therefore if the object is aliased, this property states that access to all the aliases is restricted by the lock-set of this object. The identities of these locks are not specified. This notation allows one to specify locking behavior, while hiding the details of locks involved. Verification of `\lock_protected(x.f)` would use the `\monitors_for` clause for `f` in `x`'s class.

4.2 Heap Restriction Notations

In addition to locking, thread safety can also be achieved by restrictions on references. JML's heap restriction notations are aimed at specifying how local

```
public class BetterLinkedQueue {
    protected /*@ spec_public non_null rep @*/ LinkedNode head;
    protected /*@ spec_public non_null rep @*/ LinkedNode last;
    //@ public invariant head.value == null;

    /*@ public normal_behavior
      @   requires head == last;
      @   locks this, head;
      @   assignable \nothing;
      @   ensures \result == null;
      @ also public normal_behavior
      @   requires head != last;
      @   locks this, head;
      @   assignable head, head.next.value;
      @   ensures head == \old(head.next) && \result == \old(head.next.value);
      @*/
    public /*@ atomic @*/ synchronized /*@ readonly @*/ Object extract() {
        synchronized (head) {
            /*@ readonly @*/ Object x = null;
            /*@ rep @*/ LinkedNode first = head.next;
            if (first != null) {
                x = first.value;
                first.value = null;
                head = first;
            }
            return x;
        }
    }
```

Fig. 5. Extended JML specification for `extract()`

variables may refer to objects in the heap, and how these objects may refer to each other. They allow dealing with issues like representation exposure [16] and other kinds of unwanted aliasing that would otherwise prevent modularly checking atomicity specifications. For example, consider the method `extract()` in Fig. 1. This method accesses the field `head` by first acquiring the lock on the object, so as to ensure atomicity. However, if there is representation exposure, in particular if there is another reference to the object pointed to by `head`, then that alias might be held by another thread. Thus one would have to examine other code in the program to rule out access by some other thread to the state of the object `head` refers to, in particular to the field `head.next`. In other words, representation exposure of this sort would necessitate a non-local analysis of the program to rule out such possible interference.

To prevent these problems we take advantage of the Universe type system [17], an ownership type system that already exists in an experimental form in JML [23]. This type system adds the modifiers `rep` and `readonly` to declarations.

The `rep` modifier can be used on field declarations. It states that the object referenced by the specified field is part of the representation of the given class. There can be no references from outside an object of the class to such representation objects. From outside the class, one can only refer to the enclosing object, which is the *owner* of the representation objects. For example, in Fig. 5 the fields `head` and `last` are `rep` fields, therefore there can be no external aliases to the objects to which these fields refer, and hence no representation exposure. This enables the modular verification of the atomicity specification.

```
/*@ normal_behavior
  @   requires c != null && \thread_local(c);
  @   assignable elementCount, elementData;
  @   ensures elementCount == c.size() && \fresh(elementData);
  @also
  @ exceptional_behavior
  @   requires c == null;
  @   assignable \nothing;
  @   signals (Exception e) e instanceof NullPointerException;
  @*/
public /*@ atomic @*/ Vector(Collection c) {
  elementCount = c.size();
  elementData = new Object[(int)Math.min((elementCount*110L)/100,Integer.MAX_VALUE)];
  c.toArray(elementData);
}
```

Fig. 6. Extended JML specification for a constructor in `java.util.Vector`

The `readonly` modifier is a type modifier. It marks a reference as read-only, meaning that the object cannot be modified through that reference. Read-only references are not necessarily owned by an object containing the `readonly` field, and it is often the case that such references are aliased externally. The idea is that only the identity of a `readonly` object matters to the abstract state of the enclosing object. (JML will eventually enforce various restrictions on access to read-only objects in assertions.)

The notations just discussed deal with ownership between objects. Equally important in concurrent programs is the ownership of object by threads. An *object o is owned by a thread t* if only thread t can reach o by a reference chain. This condition guarantees that there cannot be a race condition on o, because o is not shared. We introduce the notation `\thread_local(`o`)` with the meaning that o is owned by the current thread. The general syntax is as follows.

⟨thread-local-expression⟩ ::= `\thread_local (` ⟨store-ref⟩ `)`

This notation is useful for modular verification of atomicity, because accesses to thread local objects are independent (non-interfering).

For example, consider the constructor from Java's `Vector` class shown in Fig. 6. In general, constructors are independent because the constructed object is not reachable from any other thread. However, if the constructor takes object arguments to initialize the internal state of the constructed object, then its execution might not be atomic. The problem is that such an argument object might be concurrently modified by other threads. So, in this example the constructor's precondition requires that that the argument, `c`, be thread local.

4.3 Atomicity Modifier

We introduce atomicity specification into JML with a new method modifier, `atomic`. This specifies that, when a method is invoked in a state that meets its precondition, its implementation must ensure that the resulting execution is serializable. This modifier is inherited by overriding methods. Fig. 5 shows how we use this new modifier to specify `extract()` from Fig. 1.

```
public class ArrayBlockingQueue<E> {
  private /*@ spec_public non_null rep @*/ final E[] items;
  //@ monitors_for items <- lock;
  private /*@ spec_public rep @*/ final ReentrantLock lock;

  /*@ normal_behavior
    @   requires lock.isLocked() && 0 < i && i < items.length;
    @   ensures \result == \old((i + 1) % items.length) && \independent;
    @*/
  final /*@ atomic @*/ int inc(int i) {
    return (++i == items.length)? 0 : i;
  }
```

Fig. 7. Extended JML specification for `inc()`

Checking that a method declared to be atomic is actually atomic can be done in a variety of ways. For example, one could prove that the code is reducible by Lipton's theory [20, 24] or by using the notion of independent transitions. Another technique is used in the Atomizer [25], which dynamically checks that lock acquisitions and releases are properly nested and that all accesses to shared data is lock protected. The `monitors_for` clause would be used to determine what locks protect what pieces of data.

By imposing an additional obligation to guarantee serializable executions on a method's implementation, the `atomic` modifier simplifies the implementation's proof of functional correctness. The functional correctness proof can assume that the execution is serializable, thus avoiding internal interference. For example, in the method `extract()` specified in Fig. 5, the postcondition of each specification case must hold only for traces in which the method is executed sequentially. However, when combined with the additional proof of atomicity, one still gets a strong correctness guarantee about the complete behavior of the implementation. This division of proof obligations for implementations allows proofs of functional correctness to be separated from synchronization details.

From the caller's point of view, there is no potential for internal interference by atomic methods, and thus the caller only has to worry about external interference. To avoid external interference, the caller must ensure that objects needed to preserve the truth of any precondition or postcondition are thread safe (*e.g.*, locked or local to the caller's thread [22]). This additional requirement helps shake out synchronization bugs without heavyweight temporal logic—this is something that has not been explored in other static/dynamic analyses for atomicity.

It is also possible for an atomic method to transfer some of its obligation to ensure atomic execution to the caller, by stating a precondition involving locks or thread ownership. For example, an atomic method can require some locks to be held before being called, using a precondition such as `\lockset.has(lock)`, which says that the current thread holds the lock named `lock`. Such a precondition may have the added benefit of preventing external interference. Indeed, having the ultimate clients obtain locks may be sensible from an overall design standpoint (via an end-to-end argument [26]). Figure 7 show an example of this case, in which the method `inc()` transfers the responsibility of holding the lock on `lock` to the caller. (This ability to transfer some obligation to the caller shows how `atomic` is different than Java's `synchronized` modifier.)

Finally, in many concurrent classes, all methods should be atomic. To allow a designer to state this, the `atomic` keyword can be used in a class or interface declaration. Such a modifier simply states that all the methods declared in the type's declaration are atomic. This type modifier is inherited by subtypes.

4.4 Independent Predicate

A method execution is *independent* if all of its transitions are independent. For example, accesses to objects local to the thread and accesses to objects that are protected by locks are all independent transitions, since the other threads cannot observe such accesses. To specify this property of a method execution, we introduce a new specification predicate, `\independent`. This predicate can only be used in postconditions.

An example that shows how independence can be used to avoid external interference is `java.util.concurrent.ArrayBlockingQueue`'s method `inc()`. This method is specified in Fig. 7. The precondition states that the method must be called from a context in which `lock` is locked. Since `items` is protected by `lock`, due to the `monitors_for` declaration, and since `i` is a parameter of the method, no other thread can access the data used by `inc()` and cause any interference. Thus, when the precondition is met, this method's executions are independent. Furthermore, since the caller must hold the lock, and since the result is not accessible to other threads, calls cannot suffer external interference.

4.5 Blocking Behavior and Commit Atomicity

In this section we describe notations for handling methods that wait for some condition to become true before they proceed to take some (atomic) action. Such methods have what we call a *blocking behavior*.

Consider the method `take()`, from `ArrayBlockingQueue` in Java 1.5, as shown in Fig. 8. This method takes an element from the queue, but if the list is empty, it waits until there is an element to remove.

To specify the blocking behavior of methods, we use JML's `when` clause (adapted from Lerner's work [14]):

$\langle$when-clause$\rangle$ ::= `when` $\langle$predicate$\rangle$;

Its meaning is that, if a method is called in a state in which the method's `when` predicate does not hold, the method blocks until this predicate is satisfied (presumably by an action of a concurrent thread). This specification does not constrain what protocol is used to wait for the condition to become true; for example, a busy wait loop might be used. A blocking method specification can be formalized as a partial action that does not execute until the `when` predicate holds, and then atomically transitions from the pre-state to the post-state (as specified by the pre- and postcondition, *etc*). The `when` clause by default has a value of `true` (for JML's heavyweight specification cases).

```
/*@ public normal_behavior
  @   locks this.lock;
  @   when count != 0;
  @   assignable items[takeIndex], takeIndex, count;
  @   ensures \result == \old(items[takeIndex]) && takeIndex == \old(takeIndex + 1)
  @        && count == \old(count - 1);
  @*/
public /*@ atomic @*/ E take() throws InterruptedException {
  final ReentrantLock lock = this.lock;
  lock.lockInterruptibly();
  try {
    try {
      while (count == 0)
        notEmpty.await();
    } catch (InterruptedException ie) {
      notEmpty.signal(); // propagate to non-interrupted thread
      throw ie;
    }
    /*@ commit: @*/ E x = extract();
    return x;
  } finally {
    lock.unlock();
  }
}
```

Fig. 8. Extended JML specification for `take()`

To check `when` clauses, we use a special statement label, `commit`. If this label is not present in a method, it is implicitly assumed at the method body's end. This label gives a *commit point* for the method; when execution reaches the commit point, the method is no longer blocked and the rest of the method's execution must be atomic. Also, the predicate given in the `when` clause must hold at the commit point, but not necessarily during the rest of the method's execution. This idea is related to the concepts of commitment in database transactions and the notion of "commit atomicity" introduced by Flanagan [27].

Figure 8 illustrates the use of the `when` clause and the `commit` label. Method `take()` removes an element from the queue, and blocks if the queue is empty. The `when` clause in Fig. 8 says that the method may proceed only when `count` is not zero. The commit point of this method is where the `commit` label appears, right after the loop that blocks until the queue is non-empty.

4.6 Notations for Lock Types

Another important set of notations has to do with identifying what classes and interfaces have instances that are intended to be locks. Such lock objects are an addition to the implicit reentrant lock in each object that can be acquired using Java's `synchronized` statement. Java 1.5 adds several such types of lock objects, which support new concurrency patterns. An example is the new class `ReentrantLock`, whose instances are specialized locks that are manipulated by method calls. Such locks can make synchronization more flexible and allow for more efficient code. In JDK 1.5, users can also define their own locks by implementing the interface `java.util.concurrent.locks.Lock`.

To deal with these new kinds of locks, JML will consider a type to be *lock type* if it is a subtype of the `Lock` interface. An expression whose static type is a lock type is considered to denote a *lock object*. There is a potential semantic ambiguity that arises because lock objects also contain Java's implicit synchronization locks. To resolve this ambiguity we assume that when lock objects are mentioned in a context where a lock is expected, the specifier always means the lock object itself, not the implicit synchronization lock it contains. For example, in a `monitors_for` clause a lock object expression refers to the lock object itself.

To know when a lock object is locked, we introduce a new type-level declaration, the `locked_if` clause. Its syntax is as follows:

⟨locked-if-clause⟩ ::= `locked_if` ⟨predicate⟩ `;`

Each type that is a subtype of `java.util.concurrent.locks.Lock` must declare or inherit exactly one `locked_if` clause. This clause states a predicate that holds if and only if the given instance of the lock type is in the *locked* state.

So, for example, for the class `ReentrantLock` we have:

```
package java.util.concurrent;
import java.util.concurrent.locks.Lock;
public class ReentrantLock implements Lock, java.io.Serializable {
   //@ locked_if isLocked();
   /* ... */
}
```

In `ReentrantLock`, `isLocked` is a (pure) method that returns `true` if the target instance is locked, and `false` otherwise. The `locked_if` clause is used in the `lockset()` operator's semantics. For example, if `rl` has type `ReentrantLock`, then `\lockset().has(rl)` returns `true` if and only if `rl.isLocked()` holds.

4.7 Revisiting External Interference

In Sec. 4.3 we described how the `atomic` modifier solves the internal interference problem by providing an abstraction in which other threads did not interleave during the execution of the method. Now we look at how the rest of the annotations help in solving the problem of external interference. As suggested in Sec. 4.3, the key to preventing this problem is disallowing access from other threads to the objects mentioned in the pre- and postconditions.

External interference is a problem of interference between two methods: one method (the caller) calling another method (the callee) and other threads breaking the contract between the two of them. To support contract specifications that account for external interference we consider two cases: an atomic method calling another atomic method, and a non-atomic method calling an atomic method. Note that atomic methods can only call atomic methods, and the case where a non-atomic method calls another non-atomic method, aside from being uncommon, would not be handled by our notations.

In the first case, if an atomic method calls another atomic method, then the interference between the caller and the callee would be internal interference in the caller, which is already handled by the atomicity abstraction.

In the second case, when a non-atomic method calls an atomic method, the caller needs to ensure that the objects needed to preserve the truth of the pre-

and postcondition are *thread safe*. We define thread safety by introducing a new operator, `\thread_safe`, defined such that

`\thread_safe`(SR) $\equiv$ `\thread_local`(SR) `||` `\lock_protected`(SR).

That is, SR is thread-safe if it is owned by the current thread or is lock protected. To avoid external interference, a contract must require that all objects needed to preserve the truth of the pre- and postcondition are thread-safe. While this is a strong condition, in our experience it is satisfied by all well-written multi-threaded code.

As an example of thread safety let us take another look at Fig. 6. In that example, the specification requires the collection `c` to be thread local, however this condition is actually stronger than what is actually needed. The actual requirement is that the collection be free of interference. Therefore, we can relax the precondition in the `Vector` constructor to `\thread_safe(c)`, accounting for the instances in which the constructor is called with an argument that is externally protected by a lock.

5 Evaluation

In this section we describe our experiences in applying our JML extensions. We evaluate their adequacy and efficacy by applying them in a collection of *specification case studies*. In these studies, we attempt to write complete behavioral specifications for a collection of Java classes drawn from the literature that were designed explicitly for use in multi-threaded programs. These classes use significantly more complex concurrency policies than do typical classes, *e.g.*, Java container classes. Thus, if we can support the specification of rich functional properties for these classes, then our extensions will be broadly applicable. We also evaluate the checkability of our JML extensions in a set of *verification case studies*. In these studies, we sampled the classes and specifications from our specification studies and checked them using an extension of the Bogor model checking framework [28] described below.

The next two sections present the details of these of case studies, give a summary of the results obtained, and an account of our conclusions. The complete set of artifacts used in our studies are available from the web [7].

5.1 Specification Case Studies

To assess the adequacy and behavior coverage of the extensions to JML, we identified a set of of concurrent Java classes and wrote specifications for their methods using the extended JML. The classes come from multiple sources and most are implementations of concurrent data structures:

– A bounded buffer, `BoundedBuffer` (from Hartley [29]).
– Dining philosophers, `DiningPhilosophers` (from Hartley [29]).
– A linked queue, `LinkedQueue` (from Lea [12]).

Table 1. Summary of statistics from specification case studies with the extended JML. The classes marked with a * belong to Java 1.5's package `java.util.concurrent`

Class Name	Number of methods	Frequency of annotations				
		`atomic`	`\independent`	`locks`	`\thread_safe`	`when`
`BoundedBuffer`	3	3	0	2	0	2
`DiningPhilosphers`	7	7	4	2	0	1
`LinkedQueue`	7	7	0	7	0	1
`RWVSN`	8	8	2	4	0	2
`java.util.Vector`	45	45	4	34	9	0
`ArrayBlockingQueue`*	19	19	7	15	3	2
`CopyOnWriteArrayList`*	27	27	6	13	12	0
`CopyOnWriteArraySet`*	13	13	2	6	5	0
`DelayQueue`*	17	17	3	14	4	2
`LinkedBlockingQueue`*	17	17	4	12	1	2
`PriorityBlockingQueue`*	21	21	4	10	1	1
`ConcurrentLinkedQueue`*	11	11	2	0	2	4
Total:	**195**	**195**	**38**	**119**	**37**	**17**

– Code for readers-writers, `RWVSN` (from Lea [12]).
– The class `java.util.Vector`.
– Eight concurrent classes from `java.util.concurrent` in Java 1.5.

The 8 classes from `java.util.concurrent` are particularly important, as they have fairly complex and varied concurrency patterns and represent the new Java concurrency paradigm.

Table 1 presents statistics on the specifications we developed. The data shown is only for the 195 public methods in the studied classes; including private methods brings the total to over 220. We note that for all methods we were able to write complete behavioral specifications. So, for this challenging set of concurrent classes, our extensions appear sufficient for capturing their behavior.

Table 1 also reports the frequency with which we used different groups of extended JML primitives; these groups were described in the sub-sections of Sec. 4. Each entry shows the number of methods in the class whose specification used an annotation in the given group.

We observe that all of the methods studied had specifications that used the keyword `atomic`, that is, the methods exhibit the atomicity property. These results add to existing evidence [25] in support of the conclusion that most Java methods are intended to execute atomically. This validates our approach to using atomicity as the central abstraction for extending JML to support concurrency.

The use of `\independent` is not particularly common in this collection of classes. We believe that this is due to the fact that methods in these classes generally have complex concurrency policies. More typical classes with *get* and *set* methods for instance fields would probably yield large numbers of independent methods, but a broader study of Java classes is needed to confirm this intuition.

The study confirms the popularity of synchronization in enforcing correct thread-safe class behavior as more than 60% of the methods used the locking extensions. Use of data confinement is much less common in this study with less than 20% of the methods using `\thread_safe` annotations.

Table 2. Summary of statistics from verification case studies with the extended JML. Classes marked with a * belong to Java 1.5's package `java.util.concurrent`

Class Name	Number of Methods	Checkable Atomicity	Checkable Functionality	Coverage Ratio
`BoundedBuffer`	3	1	3	.67
`DiningPhilosphers`	7	6	7	.93
`LinkedQueue`	7	1	7	.57
`RWVSN`	8	4	8	.75
`CopyOnWriteArrayList`*	10	5	10	.75
`LinkedBlockingQueue`*	7	3	7	.71
Total:	**42**	**20**	**42**	**.74**

We believe that the sparse use of `when` clause in this study is due to the fact that most of our classes are container data structures. In most cases, concurrent data structures have two blocking methods: one that inserts elements but blocks if the structure is full and another one that removes elements but blocks if the structure is empty. More varied interfaces for accessing and modifying stored data will increase the need for this annotation.

The most important result of these studies is the fact that the proposed JML extensions appear to be both necessary (all annotations are used in the study) and sufficient (all methods in the study could be specified) for supporting thread-safe functional specification.

5.2 Verification Case Studies

In previous work [28], we showed how an extensible model checking framework, called Bogor, could be extended to check complex JML specifications [8], and how that framework could be independently extended to check atomicity specifications [9]. We have integrated these two separate extensions to execute simultaneously during state-space analysis to check extended JML concurrency specifications. The main technical novelty of this integration is that postcondition and frame condition checking is enforced only if the current execution of the method was serial, that is, if the method body was executed without any interleaving from other threads. So this strategy both checks the functional specifications and independently assures that all concurrent runs of the method conform to the atomicity specifications. If either the atomicity specification or the functional specification are not satisfied, then Bogor reports a specification violation.

The result of applying this specification checking tool to a subset of the classes listed in Table 1 are summarized in Table 2. The first column in this table displays the class name for the particular case study. The second column shows the total number of methods involved in the case study. For some classes, only a fraction of the total methods in the class were checked. We selected methods with diverse functionality instead of checking large numbers of similar methods.

The rest of the columns in the table present data on the degree to which the tool was capable of reasoning about specified methods. We divided the specification into two parts: the atomicity specification and the functional specification. The third column in the table shows the number of methods in the class for which the atomicity specification could be checked, and the next column shows the number of those for which the functional specification could be checked. Finally, the last column gives a ratio of specifications checked versus total specifications written for all methods in a class.

Table 2 shows that the tool could verify all of the functional specifications for each method in the study. We note that these are strong specifications that involve quantification over heap elements, checking frame conditions, freshness, reachability, and calculating the values of memory locations in the pre-state.

Checking of atomicity is not nearly as complete. The tool could verify atomicity for only 20 out of the 42 methods. The study included 22 methods that exhibit a kind of atomicity which Bogor cannot verify. Bogor's atomicity checking mechanism is based on Lipton's reduction [19] and transition independence [9], whereas the 22 uncovered methods in these case studies exhibit a different type of atomicity. 11 of those 22 methods exhibit *commit atomicity* as defined by Flanagan in [27]. In that work, Flanagan described a model checking algorithm that allows checking commit atomicity specifications. This technique could be integrated into Bogor, and would yield a coverage ratio of .87.

The other 11 uncheckable methods implement complex concurrency patterns that our tool could not detect, even if enhanced to detect commit atomicity. The model checker could be further extended, of course, to include these synchronization patterns and thereby increase checking coverage. But since checking atomicity is undecidable in general, there will always be some patterns that the tool could not detect. Fortunately, the complex patterns and challenging concurrency classes that we selected for our study are not common in real application code. Indeed, Flanagan and Freund found that more than 90% of the methods they analyzed [25] exhibited relatively simple forms of atomicity. Thus we expect that many real programs specified with extended JML will be amenable to analysis via model checking. However, a significantly broader evaluation of the use of JML and its support for analysis will be needed to confirm this conjecture.

6 Related Work

Perhaps the closest related work to ours is the work on extending Spec# to deal with multi-threaded programs [30]. The specification language part of Spec# [5], is similar in many ways to JML, although it is integrated into the programming language (as in Eiffel [6]). Like JML, Spec# also has an extensive tool set, including runtime assertion checking and a verification engine. Although Spec# is very similar to C#, it is a new programming language that extends and modifies C# in several ways. The most interesting of these changes to C# come in the ways that Spec# deals with alias control and concurrency control. In both of these areas, Spec# uses new statements (`pack` and `unpack` for alias control, and `acquire` and `release` statements for concurrency control). The treatment of alias control is more dynamic than that found in the Universe type system which JML uses, which may make it more difficult to analyze statically. For concurrency control, Spec# deals with external interference in a drastic fashion, by having `acquire` gain exclusive access to an object, so that it is thread local. The Spec# discipline solves the internal and external interference problems, and has a proof of soundness. However, the approach only applies to programs that can be written following that discipline. The authors list as future work "extending the approach to deal with other design patterns" [30–Sec. 9]. In contrast, our work attempts to deal with existing concurrent Java programs, without requiring that they follow a particular programming discipline.

Ábrahám *et al.* [31] provide a proof system for multi-threaded Java programs. Their analysis is sound and tool supported. However, as they rely on whole-program Owicki-Gries style annotations they do not achieve modularity in the sense we aim for (*i.e.*, at the level of individual compilation units). Furthermore, their proof system only deals with monitor synchronization, whereas our approach is applicable to all Java, and accepts a very wide range of synchronization patterns by abstracting away from synchronization conditions. Thus our approach promises to be more useful for existing Java code.

Robby et al. [8] identified the problem of *internal interference* described in Sec. 2. They solved it by refactoring the functional code of a method, into another method, separating it from the synchronization code. In this way they are able to check JML specifications upon the refactored method that is always called within an atomic context. However, this technique is both limited in its applicability and inconvenient for users.

Freund and Qadeer implement a modular analysis for atomic specifications on multi-threaded software [32]. The idea of using a label to mark the commit points of a method, similar to the `commit` label introduced in Sec. 4.5, comes from their work. They achieve modularity by annotating shared variables with an access predicate, and by using the concepts of reduction to link a procedure to its specification. They translate a multi-threaded program into a sequential program in which atomic procedures are executed sequentially. However, JML is more expressive than the specification language they used.

Hatcliff et al. [9] developed a technique to verify atomicity annotations using model checking. Wang and Stoller [33] provide two atomicity detection algorithms based on runtime analysis: one based on Lipton's reduction [19] and another based on a sophisticated pattern matching mechanism. However, this system only provides verification of atomicity specifications. The verification tool described in Sec. 5 can be viewed as a natural extension to these techniques, which also checks functional specifications.

7 Conclusions and Future Work

We have extended JML by adding notations that allow the verification of multi-threaded Java programs. The overall approach is to use the concept of atomicity, from Lipton's reduction theory for parallel programs. We have shown how the added annotations support the concept of atomicity and allow the specification of locking behavior. In addition, we have shown how the concept of atomicity can be used to avoid the problems of internal and external interference, and thus to support modular reasoning. We have described our success in writing extended JML specifications of existing Java classes and have reported results on the implementation of a tool that leverages these language extensions to verify behavioral specifications of multi-threaded programs.

We are planning on extending and continuing this work along several lines. On the JML language side, we are planning to work on a formalization of all the new language constructs presented in this paper, and introduce a formal modular analysis for behavioral specifications of multi-threaded programs. Some details, such as how to extend JML's concept of pure methods to allow for locking [3] also need to be worked out. Also, we are studying other ways to improve concurrency support in JML. For example, one way in which JML could be further improved is the addition of temporal logic specification operators based on specification patterns [34] as in BSL (Bandera Specification Language) [35]. There are synchronization pattern implementations, such as those presented in [36], for which it is not clear whether the extensions presented in this work are sufficient, and that might require JML to be extended with temporal logic annotations to be properly specified.

On the tool support side, the same basic division of labor described in Sec. 5 for model checking, could be used to adapt JML's runtime assertion checking tool [37] to our JML extensions. This tool instruments Java programs with additional instructions that check method pre- and postconditions, invariants, *etc.* The idea would be to add checks from the Atomizer tool [25], which checks that program traces conform to Lipton's atomicity pattern. By separately checking for atomic executions, the runtime assertion checker could carry on as before, assuming that atomic methods were executed sequentially.

We plan to integrate this work into the JMLEclipse framework [38] which is an Eclipse-based front-end for JML verification engines (in particular, it will be the front-end for our JML model checking tool). Another possible path for future work is to extend other JML tools, such as ESC/Java2 or other verification tools (*e.g.*, [39]) to incorporate the new features.

Acknowledgments

This work was supported in part by: the U.S. Army Research Office (under grant DAAD190110564), by DARPA/IXO's PCES program (AFRL Contract F33615-00-C-3044), by NSF (CCF-0306607, CCF-0341179, CCF-0428078, CCF-0429149, CCF-0429567, CCF-044167), and Lockheed Martin. Thanks to Curtis Clifton, David Cok, Suresh Jagannathan, Joe Kiniry, Rustan Leino, Todd Wallentine and the ECOOP program committee for comments on earlier drafts.

References

1. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer (STTT) (2004) To appear.
2. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06y, Iowa State University, Department of Computer Science (2004) See www.jmlspecs.org.
3. Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the design of JML accommodates both runtime assertion checking and formal verification. Science of Computer Programming **55** (2005) 185–208
4. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D.R., Kiniry, J.: Jml reference manual. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org` (2005)
5. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Proceedings of the International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices. (2004) To appear.
6. Meyer, B.: Object-oriented Software Construction. Second edn. Prentice Hall, New York, NY (1997)
7. SAnToS: SpEx Website. |http://spex.projects.cis.ksu.edu— (2003)
8. Robby, Rodríguez, E., Dwyer, M., Hatcliff, J.: Checking strong specifications using an extensible software model checking framework. In: Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Volume 2988 of Lecture Notes in Computer Science., Springer (2004) 404–420
9. Hatcliff, J., Robby, Dwyer, M.: Verifying atomicity specifications for concurrent object oriented software using model checking. In: Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation. Volume 2937 of Lecture Notes in Computer Science., Springer (2004) 175–190
10. Pugh, W.: Fixing the java memory model. In: Proceedings of the ACM 1999 Conference on Java Grande, New York, NY, USA, ACM Press (1999) 89–98
11. Flanagan, C., Freund, S.N.: Type-based race detection for java. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, New York, NY, USA, ACM Press (2000) 219–232
12. Lea, D.: Concurrent Programming in Java: Second Edition. Addison-Wesley (2000)
13. Raghavan, A.D., Leavens, G.T.: Desugaring JML method specifications. Technical Report 00-03d, Iowa State University, Department of Computer Science (2003)
14. Lerner, R.A.: Specifying Objects of Concurrent Systems. PhD thesis, School of Computer Science, Carnegie Mellon University (1991) TR CMU–CS–91–131.
15. Boyland, J., Noble, J., Retert, W.: Capabilities for sharing. In: Proceedings of the 15th European Conference on Object Oriented Programming. Volume 2072 of Lecture Notes in Computer Science., Springer-Verlag (2001) 1–27
16. Noble, J., Vitek, J., Potter, J.: Flexible alias protection. In: Proceedings of the 12th European Conference on Object Oriented Programming. Volume 1445 of Lecture Notes in Computer Science., Springer-Verlag (1998) 158–185
17. Müller, P., Poetzsch-Heffter, A.: Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen (2001) Available from `www.informatik.fernuni-hagen.de/pi5/publications.html`.
18. Borgida, A., Mylopoulos, J., Reiter, R.: On the frame problem in procedure specifications. IEEE Transactions on Software Engineering **21** (1995) 785–798

19. Lipton, R.J.: Reduction: a method of proving properties of parallel programs. Communications of the ACM **18** (1975) 717–721
20. Flanagan, C., Qadeer, S.: Types for atomicity. In: Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, ACM Press (2003) 1–12
21. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (2000)
22. Dwyer, M.B., Hatcliff, J., Robby, R.Prasad, V.: Exploiting object escape and locking information in partial order reduction for concurrent object-oriented programs. Formal Methods in System Design **25** (2004) 199–240
23. Dietl, W., Müller, P.: Universes: Lightweight ownership for jml. Journal of Object Technology (2005) To appear.
24. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, ACM Press (2003) 338–349
25. Flanagan, C., Freund, S.N.: Atomizer: a dynamic atomicity checker for multithreaded programs. In: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press (2004) 256–267
26. Saltzer, J.H., Reed, D.P., Clark, D.D.: End-to-end arguments in system design. ACM Transactions on Computer Systems **2** (1984) 277–288
27. Flanagan, C.: Verifying commit-atomicity using model-checking. In: Proceedings of the 11th International SPIN Workshop on Model Checking of Software. Volume 2989 of Lecture Notes in Computer Science., Springer (2004) 252–266
28. Robby, Dwyer, M.B., Hatcliff, J.: Bogor: An extensible and highly-modular model checking framework. In: Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering. Volume 28 number 5 of SIGSOFT Softw. Eng. Notes., ACM Press (2003) 267–276
29. Hartley, S.: Concurrent Programming - The Java Programming Language. Oxford University Press (1998)
30. Jacobs, B., Leino, K.R.M., Schulte, W.: Verification of multithreaded object-oriented programs with invariants. In: Proceedings of The ACM SIGSOFT Workshop on Specification and Verification of Component Based Systems, ACM Press (2004) To appear.
31. Ábrahám, E., de Boer, F.S., de Roever, W.P., Steffen, M.: A tool-supported proof system for multithreaded java. In: Proceedings of the International Symposia on Formal Methods for Components and Objects. Volume 2852 of Lecture Notes in Computer Science., Springer (2002) 1–32
32. Freund, S.N., Qadeer, S.: Checking concise specifications for multithreaded software. Journal of Object Technology **3** (2004) 81–101
33. Wang, L., Stoller, S.D.: Run-time analysis for atomicity. In: Proceedings of the Third Workshop on Runtime Verification (RV). Volume 89(2) of Electronic Notes in Theoretical Computer Science., Elsevier (2003)
34. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In: Proceedings of the Second Workshop on Formal Methods in Software Practice. (1998) 7–15
35. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Robby: Expressing checkable properties of dynamic systems: The Bandera Specification Language. International Journal on Software Tools for Technology Transfer **4** (2002) 34–56

36. Deng, X., Dwyer, M.B., Hatcliff, J., Mizuno, M.: Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In: Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), New York, NY, USA, ACM Press (2002) 442–452
37. Cheon, Y., Leavens, G.T.: A runtime assertion checker for the Java Modeling Language (JML). In: Proceedings of The International Conference on Software Engineering Research and Practice, CSREA Press (June 2002) 322–328
38. SAnToS: JMLEclipse Website. |http://jmleclipse.projects.cis.ksu.edu— (2004)
39. Burdy, L., Requet, A., Lanet, J.L.: Java applet correctness: A developer-oriented approach. In: Proceedings of the 12th International Symposium of Formal Methods Europe. Volume 2805 of Lecture Notes in Computer Science., Springer-Verlag (2003) 422–439

Derivation and Evaluation of Concurrent Collectors

Martin T. Vechev[1], David F. Bacon[2], Perry Cheng[2], and David Grove[2]

[1] Computer Laboratory, Cambridge University,
Cambridge CB3 0FD, U.K
[2] IBM T.J. Watson Research Center,
P.O. Box 704, Yorktown Heights, NY 10598, U.S.A

Abstract. There are many algorithms for concurrent garbage collection, but they are complex to describe, verify, and implement. This has resulted in a poor understanding of the relationships between the algorithms, and has precluded systematic study and comparative evaluation. We present a single high-level, abstract concurrent garbage collection algorithm, and show how existing snapshot and incremental update collectors, can be derived from the abstract algorithm by reducing precision. We also derive a new hybrid algorithm that reduces floating garbage while terminating quickly. We have implemented a concurrent collector framework and the resulting algorithms in IBM's J9 Java virtual machine product and compared their performance in terms of space, time, and incrementality. The results show that incremental update algorithms sometimes reduce memory requirements (on 3 of 5 benchmarks) but they also sometimes take longer due to recomputation in the termination phase (on 4 of 5 benchmarks). Our new hybrid algorithm has memory requirements similar to the incremental update collectors while avoiding recomputation in the termination phase.

1 Introduction

The wide acceptance of the Java programming language has brought garbage collected languages into the mainstream. However, the use of traditional synchronous ("stop the world") garbage collection is limiting the domains into which Java and similar languages can expand. The need for concurrent garbage collection is primarily being driven by two trends: the first is increased heap sizes, which make the pauses longer and less tolerable; the second is the increase in the use of, and complexity of, real-time systems, for which even short pauses are often unacceptable. Therefore there is need for rapid improvement in various kinds of incremental and concurrent collector technology.

Unfortunately, concurrent garbage collectors are one of the more difficult concurrent programs to construct correctly. The study of concurrent collectors began with Steele [27], Dijkstra [14], and Lamport [20].

Concurrent collectors were considered paradigmatic examples of the difficulty of constructing correct concurrent algorithms. Steele's algorithm contained an error which he subsequently corrected [28], and Dijkstra's algorithm contained an error discovered and corrected by Stenning and Woodger [14]. Furthermore, some correct algorithms [9] had informal proofs that were found to contain errors [25].

These problems also manifest themselves in practice because concurrent bugs generally have a non-deterministic effect on the system and are non-repeatable, so that connecting the cause of the error to the observed effect is particularly difficult.

Many incremental and concurrent algorithms have been introduced in the last 30 years [1, 3, 4, 6, 7, 10, 11, 12, 13, 16, 17, 18, 19, 21, 23, 24], but there has been very little comparative evaluation of the properties of the different algorithms due to the complexity of implementing even one algorithm correctly. As noted in [2], because of these constraints, current state-of-the-art concurrent systems are generally not quantitatively compared against each other and the exact relationships among the different concurrent schemes are largely unknown.

For example, early collectors were all examples of *incremental update* collectors which "chase down" modifications to the object graph that are made by the program during collection. Yuasa [29] introduced snapshot collectors, which do not attempt to collect garbage allocated after collection begins, but do not require any rescanning of the object graph. Thus, snapshot collectors trade off reliable termination for a potential increase in floating garbage. However, costs and benefits relative to incremental update techniques have not been systematically studied.

This paper presents a high-level algorithm for concurrent collection that subsumes and generalizes several previous concurrent collector techniques. This algorithm is significantly more precise than previous algorithms (at the expense of constant-factor increases in both time and space), and more importantly yields a number of insights into the operation of concurrent collection. For instance, the operation of concurrent write barriers can be viewed as a form of degenerate reference counting; in our algorithm, we do true reference counting and are thereby able to find live data more precisely.

Existing algorithms can then be viewed as instantiations of the generalized algorithm that sacrifice precision for compactness of object representation and speed of the collector operations (especially the write barriers).

Additionally, we argue that all of the existing concurrent algorithms fundamentally share a deeper structure. And there is a whole continuum of existing algorithms, which we have not yet explored, but could be uncovered if we start from such a structure. Moreover, by having a common abstract algorithm, much of the construction of the practical collector will be simplified.

The contributions of this paper are:

– A generalized, extendable, abstract concurrent collection algorithm, which is more precise than previous algorithms;
– A demonstration of how the abstract algorithm can be instantiated to yield existing snapshot and incremental update algorithms;
– A new snapshot algorithm (derived from the abstract algorithm) that allocates objects unmarked ("white") and reduces floating garbage without re-scanning of the heap required by incremental update algorithms;
– An implementation of four concurrent collectors in a production-quality virtual machine (IBM's J9 JVM product): Snapshot (after Yuasa), two incremental- update (after both Dijkstra and Steele), and our hybrid snapshot algorithm; and
– A quantitative experimental evaluation comparing the performance of the different algorithms.

2 An Abstract Collector

This section presents the abstract collector algorithm. The algorithm is designed for maximum precision and flexibility, and keeps much more information per object than would be practical in a realistic implementation. However, the space overhead is only a constant factor, and thus, does not affect the asymptotic complexity of the algorithm, while the additional information allows a potential reduction in complexity.

Similarly, a number of operations employed by the abstract algorithm also have constant time overheads that would be undesirable in a realistic collector. In particular, there is no special treatment of stack variables: they are assumed to be part of the heap and therefore every stack operation may incur a constant-time overhead for the collector to execute an associated barrier operation. There are a number of collectors for functional languages (such as ML and Haskell) that treat the stack in exactly this way.

Our generalized concurrent collection algorithm makes use of the framework of Bacon et al. [5]: they showed that for synchronous ("stop the world") garbage collection, tracing and reference counting can be considered as dual approaches to computing the reference count of an object. Tracing computes a least fixpoint, and reference counting computes a greatest fixpoint. The difference between the greatest and least fixpoints is the cyclic garbage. In most practical tracing collectors, the reference count is collapsed into a single bit.

Furthermore, they showed that all collectors could be considered as a combination of tracing and reference counting, and that any incrementality is due to the use of a reference counting approach with its write barriers.

This insight is now extended to concurrent tracing collectors: we show that they are also a tracing/reference counting hybrid. The collector traces the original object graph as it existed at the time when collection started, but does reference counting for pointers to live objects that could be lost due to concurrent mutation.

The abstract algorithm makes use of the variables depicted in Table 1. In the discussion that follows, we elaborate more on the semantics of each shared variable.

Table 1. Shared variables, $|N|$ is object size, $|P|$ is maximum number of pointers in the heap and $|H|$ is the number of objects in the heap

Shared Variable	Description	Computed By	Value Domain
Global Variables			
Phase	Current Collector phase	Collector	[Idle, Tracing, Sweeping]
Hue	Scanned Part of the Heap	Collector	$[0, \|H\|]$
Per-Object Variables			
Marked	Mark flag	Collector	Boolean
SRC	Scanned reference count	Mutator	$[0, \|P\|]$
Shade	Scanning progress within object	Collector	$[0, \|N\|]$
Recorded	Recorded in buffer by barrier	Mutator	Boolean
DontSweep	Allocated after Hue	Mutator	Boolean

2.1 Restrictions and Assumptions

The algorithms we discuss are non-moving and concurrent, but not parallel. That is, the collector is single-threaded. The ideas derived from this discussion, however, are easily extendable to algorithms using multiple spaces, such as generational ones.

Furthermore, the algorithm performs synchronization with atomic sections rather than isolated atomic (compare-and-swap) operations. Atomic sections are relatively expensive on a multiprocessor, so that although the algorithm can be executed on a multiprocessor it is better suited to a uniprocessor system based on safe points, in which low-level atomicity is a by-product of the implementation style of the run-time system.

Additionally, we assume that the concurrency between the mutators and the collector is bounded by a single cycle. This is a common underlying assumption in most practical algorithms. Essentially, this means that all mutator operations started in collector cycle N finish in that cycle. They do not carry over to cycle N + 1, for example. No pipelining between the collector phases is assumed: sweeping is followed by marking.

For the sake of presentation, we also make a number of simplifying assumptions about the heap. We assume that all heap objects are the same size S and consist only of collector meta-data and object data fields which are all pointers. The fields of an object X are denoted $X[1]$ through $X[S]$.

2.2 Tracing

The abstract algorithm is shown in Fig 1 and 2. We begin by describing the outer collection loop and the tracing phase of collection cycle.

The `Collect()` procedure is invoked to perform a (concurrent) garbage collection. When it starts, the `Phase` of the collector is `Idle`, and the first thing it does is to atomically mark the root object and set the collector phase to `Tracing`. Atomicity is required because mutators can perform operations dependent on the collection phase.

Because all variables live in the heap, there is only a single root that must be marked atomically. In a realistic collector that avoided write barriers on stack writes, this single operation would be replaced by atomic marking of all of the roots – which could be on stacks or on global variables.

The core of the algorithm is the invocation of `Trace()`, which is performed repeatedly until the concurrently executing mutators have not modified the object graph in a way that could result in unmarked live objects.

Tracing in our algorithm is very similar to the tracing in a synchronous collector: it repeatedly gets an object from the mark stack and scans it.

Shades of Grey. In the `Scan()` procedure the first major difference appears. Like a standard tracing collector, we iterate over the fields of the object and mark them. However, as each field is read, the *Shade* of the object is incremented.

The use of shades is one of the generalizations of our algorithm. Most concurrent collectors use the well-known *tri-color* abstraction: an object is white if it has not been seen by the collector, grey if it has been seen, but all of its fields have not been seen, and black if both it and its fields have been seen.

The color of an object represents the progress of the tracing wavefront as it sweeps over the graph. However, the tri-color abstraction loses information because it does not

```
Collect()
    atomic
        Mark(root);
        Phase = Tracing;

    do
        Trace();
    while (ProcessBarriers());

    atomic
        ProcessBarriers());
        Trace();
        Phase = Sweeping;

    Sweep();
    Phase = Idle;

Trace()
    while(! markStack.empty())
        Obj = markStack.pop();
        Scan(Obj);

Scan(Obj)
    for (field = 1; field <= Obj.Size; field++)
        atomic
            Ptr = Obj[field];
            Obj.Shade = field;
        Mark(Ptr);

Mark(Obj)
    if (! Obj.Marked)
        markStack.push(Obj);
        Obj.Marked = true;

ProcessBarriers()
    retrace = false;
    atomic
       while (true)
          if (barrierBuffer.empty())  return retrace;
          Obj = barrierBuffer.remove();
          Obj.Recorded = false;
          if ((INSTALLATION_COLLECTOR && Obj.SRC == 0) ||
              (DELETION_COLLECTOR && Obj.SRC == 0 && isLeaf(Obj)))
              continue;
          if (! Obj.Marked)
            Mark(Obj);
            retrace = true;
```

Fig. 1. Abstract Collector Code

```
Sweep()
    for (i = 1; i <= Heap.Size; i++)
        Hue = i;
        Obj = Heap[i];
        if (! Obj.Marked && ! Obj.DontSweep)
            FREE(Obj)
        Reset(Obj)

    Hue = 0;

Reset(Obj)
    Obj.Shade = Obj.SRC = 0;
    Obj.Marked = Obj.Recorded = Obj.DontSweep = false;
```

```
atomic WriteBarrier(Obj, field, New, isAllocated)
    if (Phase == Tracing)
        Old = Obj[field];

        if (field < Obj.Shade) // Already scanned by collector
            if (! New.Marked)
                if (DELETION_COLLECTOR)
                    if (isAllocated)
                        Remember(New);
                else
                    if (! New.Recorded)
                        Remember(New);
                New.SRC++;
            if (! Old.Marked)
                Old.SRC--;
        else if (DELETION_COLLECTOR && ! Old.Marked
                 && ! Old.Recorded &&
                 (!isLeaf(Obj) || (isLeaf(Obj) && Old.SRC > 0)))
            Remember(Old);
    Obj[field] = New;

atomic AllocateBarrier(Obj, field, New)
    Reset(New);
    if (Phase == Sweeping)
        if (Heap.free >= Heap.Hue)
            New.DontSweep = true;
    else
        WriteBarrier(Obj, field, New, true);

Remember(Obj)
    BarrierBuffer.append(Obj);
    Obj.Recorded = true;
```

Fig. 2. Abstract Mutator Code

track the progress of sweeping within the object. Fundamentally, the synchronization between the collector and the mutator depends on whether an object being mutated has been seen yet by the collector. Therefore, by losing information about the marking progress, the precision of the algorithm is compromised.

The *Shade* of an object is simply a generalization of the tri-color abstraction: objects are still white, grey, or black, but there are many shades of grey. The shade represents the exact progress of marking within the object. When *Shade* is 0, the object is white. When it is the same as the number of fields in the object, the object is black. We will describe how the shade information is used when we present the write barrier executed by the mutator.

Once the `Scan()` procedure has updated the shade, it marks the target object. The `Mark()` procedure pushes the object onto the mark stack if it was not already marked.

2.3 Mutator Interaction

We now turn to the interaction between the mutator and collector by considering the actions of the mutator when it changes the object graph. The connectivity graph can be modified by both pointer modification and object allocation.

Write Barrier. The write barrier is depicted by the procedure `WriteBarrier()` in Fig 2. In our presentation of the algorithm, the entire write barrier is atomic. Finer-grained concurrency is possible, but is not discussed in this paper.

The write barrier takes a pointer to the object being modified, the field in the object that is being modified, the new pointer that is being stored into the object, and a flag indicating whether the new pointer refers to an object that was just allocated.

If the collector is not in its tracing phase, it simply performs the write: because it is the tracing phase that determines reachability of objects, only object graph mutations during tracing can affect reachability (object graph additions – via allocation – require some additional synchronization, which is described below).

An object can be protected either (1) when a pointer to it is stored or (2) when a pointer to it is overwritten. We call saving the pointer at 1 an *installation barrier* and saving the pointer at 2 a *deletion barrier*. The Dijkstra-style barrier is an instance of an installation barrier; the Yuasa-style barrier is an instance of a deletion barrier.

Earlier, we described our collector as a combination of tracing and reference counting. The reference counting is done in the write barrier. In particular, we keep a count of the number of references to an unmarked object from scanned portions of the heap. This is called the Scanned Reference Count or *SRC*. The *SRC* is one of the most important aspects of our abstract algorithm and allows for a number of interesting insights.

The SRC allows us to defer reachability decisions from the time of a write barrier to the time when collector tracing is finished. For example, if a pointer to an object is installed into the scanned portion of the heap, and subsequently removed from the scanned portion of the heap, then it can not possibly affect the liveness of the object.

Object Allocation. Besides pointer assignments, the mutator can also add objects to the connectivity graph. Similarly to pointer assignments, the allocation interacts with the tracing phase. In addition, allocation also interacts with the sweeping phase of the collector. This is performed in the procedure `AllocateBarrier()` in Fig 2.

In terms of reachability, if the collector is in its tracing phase, object allocation can be seen as just another pointer modification event. The main difference between allocation and pointer writes is that upon allocation we know that the new pointer is unique. We also know that the new object does not contain any outgoing pointers.

During the sweeping phase, the collector iterates over the heap, reclaims all unreachable objects and resets the state of the live objects. We assume that we can designate which parts of the heap the collector has passed indicated by the variable *Heap.Hue*. The variable is similar to *Shade*, except *Shade* is applied per object while *Hue* is applied per heap. That is, we have one *Hue* variable. Similarly to *Shade*, the variable is monotonic within the same collector cycle.

If the mutator allocates during the collector's sweeping phase, we require a mechanism to protect the object from being collected erroneously. The field *DontSweep* indicates if the object has been allocated in a part of the heap that the collector has yet to reach in its sweeping action.

2.4 Lost Object Problem

In a concurrent interleaving between the application and the collector, the program can accidentally hide pointers during collector heap marking. A mutator can store a pointer into a portion of the heap the collector has already scanned, and subsequently destroy all paths from an unscanned reachable portion of the heap to that object. The problem can be broken down into hiding directly and transitively reachable objects. For illustration purposes an object with a black color is one that the collector has marked reachable and has scanned all of its children. A white-colored object is one that the collector has not yet reached.

The sequence for *directly* hidden objects is depicted in Fig. 3. Each state of the graph is shown in time steps. In the initial state, three are objects: scanned object Y, unscanned but reachable object X and object Z which is not yet marked, but is reachable only from X via pointer *a*. In step D1, a mutator copies pointer *a* and stores it into the scanned object Y resulting in pointer *b*. In step D2, the mutator removes the only pointer to Z from an unscanned but reachable object X. The mutator is then immediately preempted by the collector and in step D3, the collector processes object X, turns it black (scanned) and assumes that its marking phase is completed. Next, in step D4, the collector starts its sweeping phase and erroneously frees object Z, although Z is reachable from Y via pointer *b*. In this case we say that object Z is *directly* hidden from the collector.

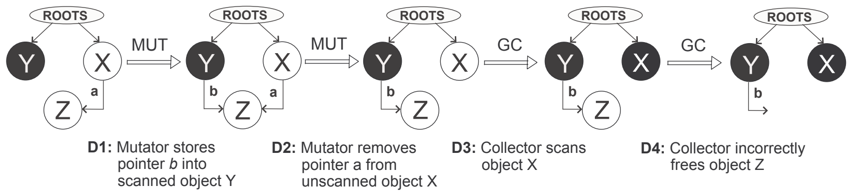

ROOTS
Y
X
Z
a
MUT
b
GC
D1: Mutator stores pointer b into scanned object Y
D2: Mutator removes pointer a from unscanned object X
D3: Collector scans object X
D4: Collector incorrectly frees object Z

Fig. 3. Erroneous collection of live object Z via deletion of direct pointer a from object X

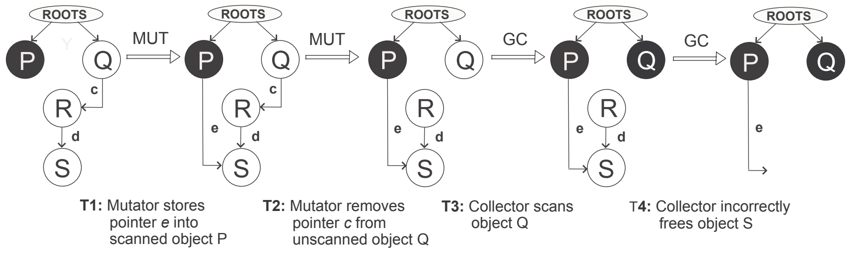

ROOTS
P
Q
R
S
c
d
e
MUT
GC
T1: Mutator stores pointer e into scanned object P
T2: Mutator removes pointer c from unscanned object Q
T3: Collector scans object Q
T4: Collector incorrectly frees object S

Fig. 4. Erroneous collection of live object S via deletion of pointer *c* from object Q which transitively reaches S through R

Alternatively, an object can be hidden *transitively*. This case is illustrated in Fig. 4. In the initial state, object P is scanned and Q, R, and S are reachable but not yet seen. Starting from this state, in step T1, the mutator introduces pointer *e* from a scanned and visited object P to object S. In step T2, the mutator destroys the unscanned pointer *c* from Q to R, essentially, destroying the only path starting from Q to object S. Next, in step T3, the collector preempts the mutator and scans object Q as shown and assumes to have finished the tracing phase. In step T4, the collector incorrectly frees object S. In this case we say that object S was *transitively* hidden from the collector.

The lost object problem consists of two main events in time: storing a pointer to the particular object to be lost and in a subsequent step destroying all other paths to that object. The two well-known solutions to this problem operate at either of these two steps. They either operate at state D1/T1 or at state D2/T2. Dijsktra's and Steele's solutions operate at states D1/T1 and aim to prevent the un-acknowledged introduction of pointers from scanned portions of the heap to reachable but unmarked objects. They essentially speculate that a pointer destruction will occur sometime in the future, and this will lead to hiding of the object. Alternatively, solutions can operate at steps D2/T2. When a pointer is destroyed as in steps D2 and T2, we reason that a pointer to the object must have been introduced earlier and make the target of the overwritten object reachable. This is the solution chosen by Yuasa. For example, Yuasa would make Z live when pointer *a* is removed in step D2 or pointer *c* is removed in step T2. In the transitive case, even though object R might have become unreachable when the pointer is destroyed in step T2, Yuasa's solution requires that object R is kept live as a potential only path left to the hidden object S.

2.5 Design Alternatives

The abstract algorithm maintains rich object and heap-level information. This section attempts to provide an intuitive understanding of the abstract algorithm.

The essence of the abstract algorithm is that it allows for deferring reachability decisions from the mutator to the collector. That is, in the write barrier the mutator detects a potential problem and nominates a candidate pointer for the collector. Subsequently, before the termination of its tracing phase, the collector examines the nominated pointers and optionally discards unnecessary candidates. The specific choices of which pointers are selected by the mutator and which pointers are processed by the collector are discussed in the following sections.

Mutator Selection. When a mutator hits the write barrier, it can protect an object using either the *installation* choice or the *deletion* choice. Intuitively, to protect an object, the mutator speculates about reachability, since it has no knowledge of how the graph changes before the collector has finished tracing. In the abstract algorithm, the mutator detects a potential problem, but does not make explicit decisions whether the object is reachable at the end of tracing.

If the *installation* choice is utilized, the object is nominated by the mutator as soon as the *SRC* becomes > 0, thereby, protecting the object *directly* rather than transitively. The installation choice speculates that right after the *SRC* becomes > 0, the only path to the object from an unscanned, but reachable object will be destroyed. Immediately after nominating the pointer, the SRC could be decremented back to zero effectively undoing the previous operation.

For the *deletion* approach, if a pointer in an unscanned object is overwritten, another object can become hidden either transitively or directly. If the *SRC(X)* is > 0 and a pointer to object X is overwritten from an unscanned portion of the heap, we need to protect object X directly. Therefore, the mutator must nominate this pointer. Alternatively, if the *SRC(X)* is 0, we might need to protect some transitively reachable object from X. The key is to recognize that if X does not contain any outgoing pointers, then no object can be hidden transitively. In such cases, we do not need to nominate X.

Determining whether object X is a leaf can be done by using the type of the object. Examples of acyclic types are scalar arrays as well as newly allocated objects before pointers are stored into them. Objects of acyclic types are leaves for their entire lifetime while newly allocated objects can be leaves only temporarily.

Moreover, even if object X is not a leaf, a write barrier could possibly perform nested checks and determine that at, for example, two-levels deep all objects pointed from X are leaves and their SRC is 0. In this case, we can again refrain from nominating the overwritten X pointer.

In some way, it would be logical to make a conclusion that the deletion choice should be more precise, since it always reasons about an event which has already occurred: the *SRC* of some object has become > 0. The *installation* choice speculates about the future, that may be at some point an unscanned pointer will be destroyed. Although a deletion collectors reasons about past event and should have more information, it has no practical way of determining those transitive objects whose *SRC* > 0. In contrast, the *installation* choice always has an immediate access to the critical object.

Besides pointer events, the mutator can modify the connectivity graph via object allocation. Allocation can be seen as an instance of a write barrier with special knowledge that the target pointer is unique. For installation choice collectors, allocation events are treated exactly as all pointer events. For deletion choice collectors, if the resulting pointer from an allocation request is stored into a scanned portion of the heap, it is possible that the object will be lost. We can then think of allocation as a normal pointer store, except that immediately after the pointer store into a scanned region of the heap, an unscanned virtual pointer to the object is overwritten. Since the virtual event cannot be captured by the barrier, we *simulate* it in the barrier. The flag *isAl-*

located is passed specifically for this reason from the `AllocateBarrier()` to the `WriteBarrier()` procedure.

Characterizing graphs that allow different barrier choices per object is an interesting though primarily theoretical question. It is generally not possible to make that decision arbitrary without some local knowledge of the graph.

Finally, if all barriers occur on leaf objects, the deletion choice will always require us to nominate fewer pointers. Of course, in both barrier choices precisely the same number of objects will be marked live. This can be logically explained by the fact that in both cases, the immediate object is available during the pointer store therefore we can reason locally about reachability. In that case, for leaf objects, the decision of which barrier to use can be made per-object rather than per-collector-cycle. We do not deal with this topic further.

Collector Choice. Once the collector has finished the initial tracing of the heap, there could be a number of unmarked candidates nominated by the mutator. It is possible that in between the time when the mutator has nominated a candidate and the collector sees it, the candidate is no longer necessary.

Similarly to the mutator's pointer selection mechanism, the collector also uses a mechanism to filter out unnecessary candidates. This selection mechanism for the collector is the same as that for the mutator. This can be seen in the write barrier processing phase, the procedure `ProcessBarriers()` in Fig. 1.

Although the collector uses the same mechanism as the mutator, it is possible that candidates nominated by the mutator are ignored by the collector. For example, if the *installation* choice is used and if the object's *SRC* is > 0, when the collector sees such pointers, the corresponding object must be retraced. If the object's *SRC* is 0 however, then the object was recorded by the mutator, but before tracing finished, its *SRC* dropped to 0. Such objects are skipped by the collector in this phase. They have either become garbage or are live but hidden. In the latter case, the object is reachable transitively from a chain of reachable objects starting at an object whose *SRC* is > 0. We therefore only need to re-trace objects whose *SRC* is > 0. Similar reasoning although with a different selection criteria is applied to the deletion choice.

Maintaining an accurate *SRC* has several advantages. First, the SRC prevents us from inducing *floating garbage*. That is because at the time a pointer store occurs, the mutator *nominates* objects that could be *potentially* hidden from the collector. It need not make an explicit decision whether they will actually be reachable once the tracing is complete. The reachability is left to the collector when the barrier tracing phase occurs. It is because of the *SRC* that the mutator does not need to make such explicit decisions about reachability. Secondly, the collector must start re-scanning only from specific objects. For example, for the installation choice it does not need to consider objects whose *SRC* is 0.

3 Transformations: Trading Precision for Efficiency

The abstract algorithm of the previous section provides a much higher degree of precision than previously published and implemented algorithms, but it is also impractical.

In this section we describe how practical collectors can be derived via orthogonal transformations of the abstract collector. Since the transformations are orthogonal, and since the reduction in precision can be modulated, this framework allows the derivation of a much broader set of algorithms than have previously been described, as we will show in the following section.

The transformations presented are (1) reduction in write barrier overhead by treating multiple pointers as roots; (2) reduction in root processing by eliminating re-scanning of the root set; (3) reduction in object space overhead and barrier time overhead by reducing the size of the scanned reference count (SRC); (4) reducing object space overhead by reducing the precision of the per-object shade; (5) conflation of shade and SRC to further reduce object space overhead and speed up the write barrier.

These transformations are not strictly semantics-preserving, since the set of collected objects is changed. However, they are invariant-preserving in that live data is never collected (the collector safety property).

3.1 Root Sets: Eliminating Write Barriers

In the abstract algorithm, all memory is reached from a single root. Thus stacks and global variables are treated as objects like any other. Such an approach is actually used in some implementations of functional programming languages [12]. However, in systems with a significant level of optimization, the cost of such an approach is prohibitive because the mutation rate of the stack is generally extremely high and every stack mutation must include a write barrier.

Therefore, we can transform an abstract algorithm with a uniform treatment of memory into an algorithm which partitions memory into two regions: the roots and the *heap*. The roots generally include the stack and may also contain the static variables and other distinguished pointer data.

In common parlance the static variables are generally considered to be roots, but if they are barriered then they are in effect treated as fields of the “global variable object”, and only the pointer to that “object” is a true root. From the point of view of the root transformation, the only issue is that the memory is partitioned into two sets, the roots and the heap, such that there are no pointers from the heap into the roots.

In the abstract collector, there is a single root pointer. Therefore, examining the root is an inherently atomic operation. With the addition of multiple roots, they must either be processed atomically or a further transformation must be applied to incrementalize root processing [15]. In this work we restrict ourselves to algorithms with atomic root processing.

In particular, at the beginning of every collection, we stop the mutators and mark all heap objects directly reachable from roots, placing them on the work queue (mark stack). Subsequently, when the roots are mutated, no write barrier is executed.

Since the roots are processed atomically at the beginning of collection, they are in effect a *scanned object*. However, since we no longer perform a barrier on mutation, the SRC field of objects referenced by mutated roots is no longer guaranteed to be correct and they will not have been placed in the barrier buffer. Therefore, the algorithm must be adjusted to correct or accommodate this imprecision.

The imprecision can be corrected by atomically *re-scanning* the roots before barrier processing. Consider a sequence of stores into a particular root pointer. These stores must be treated like stores into a scanned portion of the heap, so that the SRC of the installed and overwritten pointers must be incremented and decremented, respectively. If we rescan the roots, then any pointers which were scanned previously will have already been marked, and the SRC will be unaffected.

When a pointer to an unscanned object is stored into a root for the first time, the pointer that is being overwritten must point to a marked object, since all direct referents of roots are marked atomically at the start of collection. Thus the SRC of the overwritten pointer would not have changed if the write had been barriered. However, the SRC of the newly installed pointer would have been incremented, but if the roots are rescanned this pointer will be discovered and since it points to an unmarked object it is known to be a new pointer, and the SRC is incremented. Thus in the case of a single store to a root, the SRC is correct.

Inductively, if there are multiple stores to a root, then each subsequent store will cause the SRC of the overwritten pointer to be decremented and the SRC of the installed pointer to be incremented. The decrement will cancel the increment that was performed on the same pointer when it was previously installed. Therefore, a sequence of stores to a particular root pointer will result in the SRC of all objects except the last one to remain unchanged. [1]

Since that object is found by rescanning, rescanning will compute an accurate SRC, and the transformation that separates the memory into roots and heap leaves the precision of the algorithm unchanged.

3.2 Root Rescan Elimination

As we have just shown, the special treatment of roots does not affect the precision of the collector if root re-scanning is used to correct the SRC. However, re-scanning is undesirable because it increases the running time of the algorithm.

If root re-scanning is eliminated, then the SRC values may be under-approximations (because the increment of the final pointer stored in a root will have been missed). Since increments may keep objects live that would otherwise have been collected, this means that any reclamation of an object based on its SRC being 0 is unsafe. Therefore, the algorithm must be conservative in such cases and precision will be sacrificed.

Furthermore, when an installation barrier is used the installation of pointers into the scanned portion of the heap is what causes them to be remembered in the barrier buffer for further tracing during barrier processing. This means that regardless of the imprecision of the SRC, objects that would have had a non-zero SRC must be seen during barrier processing. In effect, this means re-scanning can not be eliminated for algorithms that use the installation barrier.

For algorithms that use the deletion barrier, the only pointers to new objects that are remembered in the write barrier are the newly allocated objects. Therefore, as long

[1] This is the same reasoning that was applied by Barth to eliminate redundant reference count updates at compile time [8], and by Levanoni and Petrank to remove redundant reference count updates between epochs in a concurrent reference counting collector [22].

as those objects are placed in the barrier buffer by the allocator, and the SRC-based computation in the barrier processing is eliminated, then the root re-scanning can be safely eliminated.

Since no collector decisions are based on the value of the SRC, it is redundant and can be eliminated. The result is an algorithm with more floating garbage (in particular, all newly allocated objects are considered live), of which Yuasa's algorithm is an example.

3.3 Shade Compression

The shade of an object represents the progress of the collector as it processes the individual pointers in the object. The precision of the shade can always safely be reduced as long as the processing of the pointers in the object in the write barrier treats the imprecise shade conservatively.

In particular, since many objects have a small number of pointers N, it is efficient to treat the shade which originally had the range $[0, N]$ as the set $\{0, [1 \ldots N-1], N\}$. These three values represent an object for which collector processing has not yet begun, is in progress, or has been completed. This is the standard tri-color abstraction introduced by Dijkstra, where the three values are called white, grey, and black, respectively.

When N is small, the chance is low that the mutator will store a pointer into the object currently being processed by the collector, so the reduction in precision is likely to be low. However, with large objects (such as pointer arrays) the reduction in precision can be more noticeable. Some collectors therefore treat sections of the array independently, in effect mapping equal-sized subsections of the array into different shades.

3.4 Scanned Reference Count Compression

The scanned reference count (SRC) can range from 0 to the number of pointers in the system. However, the number of references to an object is usually small, and the SRC will be even lower (since it only counts references from the scanned portion of the heap to unmarked objects). Therefore, the SRC can be compressed and the loss of precision is likely to be low.

However, the compression must be conservative to ensure that live objects are not collected. This is accomplished by making the SRC into a "sticky" count [26]: once it reaches its maximum value, it is never decremented. As a result, the SRC is an over-approximation, which is always safe since it will only cause additional objects to be treated as live.

An important special case for collectors that use an installation barrier is a one-bit SRC, since in this case the SRC becomes equivalent to the Recorded flag, allowing those two fields to be collapsed.

3.5 Conflation of Shade and Scanned Reference Count

In a collector using an installation barrier with a one-bit sticky SRC and tri-color shade, an object with a stuck SRC must be scanned by the collector. Similarly, a grey object must be scanned by the collector. Thus the meaning of these two states can be collapsed and the grey color can be used to indicate a non-zero (stuck) SRC, which also represents the Recorded flag.

This is in fact the representation used by most collectors that have been implemented. In effect, they have collapsed numerous independent invariants into a small number of states. This helps to understand why such algorithms are bug-prone: collapsing the states corresponding to algorithmic invariants relies on subtle transformations and simultaneously reduces redundancy in the representation.

4 Using Transformations to Derive Practical Collectors

In this section, we derive various practical algorithms by applying the previously discussed transformations to the abstract collector algorithm. Some of the schemes are well-known concurrent algorithms such as Dijkstra and Yuasa, while others are new derivations.

4.1 Derivation of a Dijkstra Algorithm

The Dijkstra algorithm is an instance of an abstract installation collector and to derive it we apply the following transformations:

1. *Root Sets* transformation
2. *Shade compression* to tri-color
3. *SRC compression* to a single sticky bit
4. *Conflation of Shade* and *SRC*

Although at the end of the transformation steps, we arrive at a practical Dijkstra algorithm, the intermediate steps also represent valid algorithms with different precision.

The compressions of *SRC* and *Shade* can lead to floating garbage. However, unlike *Shade* and *SRC* compressions, the *Conflation* transformation does not lead to increased floating garbage. On the other side, it reduces, both, space consumption in the header of the object, and, complexity of the write barrier.

The *Root Sets* transformation also preserves the treatment of allocated objects. When a new object is allocated and stored into the roots, the mutator will not nominate the pointer because the store will occur into a scanned partition (roots) and the write barrier is not active on the roots. If the pointer to the allocated object gets stored in the heap, then it will be processed in the mutator write barrier, similarly to all other objects existing at collection startup. If the allocated object dies before the roots are rescanned, the collector will not mark that object as live.

A Steele-like collector is similar to a Dijkstra collector except that its transformation covers a wider range of rescanning. A Steele algorithm is not limited to rescanning only the roots, but can also rescan heap partitions. However, the barrier processing phase and the selection criteria are exactly the same as in the Dijkstra collector.

4.2 Derivation of a Yuasa Algorithm

Our second derived collector is a Yuasa snapshot algorithm. The Yuasa algorithm is an instance of a deletion collector. The algorithm can be derived by applying the following transformations to the abstract collector:

```
MarkRoots()
    while (! roots.end())
      Obj = roots.get();
      Mark(Obj);

MarkRootsDirect()
    while (! roots.end())
      Obj = roots.get();
      if (Obj.isAllocatedInThisCycle)
          Obj.Color = black;

Collect()
    atomic
        MarkRoots();
        Phase = Tracing;

    do
        Trace();
    while (ProcessBarriers());

    atomic
        MarkRootsDirect();
        ProcessBarriers());
        Trace();
        Phase = Sweeping;

    Sweep();

    Phase = Idle;

atomic WriteBarrier(Obj, field, New)
    if (Phase == Tracing)
        Old = Obj[field];

        if (New.Color == white && New.isAllocatedInThisCycle)
            New.Color = black;

        if (   Obj.Color != black && Old.Color == white
            && !Old.Recorded )
            Remember(Old);
    Obj[field] = New;
```

Fig. 5. Pseudo Code For The Hybrid Collector

1. *Root Sets* transformation
2. *Shade compression* to tri-color
3. *Root Rescan Elimination*

During barrier processing, deletion collectors can skip objects whose *SRC* is 0 and are leafs. However, since *Root Rescan elimination* prevents the roots rescanning process, an accurate *SRC* cannot be computed, and subsequently the collector selection criteria cannot be applied. Therefore, in order to preserve the safety property of the abstract collector, the collector must mark all overwritten pointers and rescan from them. The *SRC* is removed since it cannot serve its primary purpose: a guide for the collector selection criteria.

The Yuasa collector is the most conservative approach to floating garbage. It does not allow any destruction in the connectivity graph once the collector has started and it effectively allocates only reachable object (black).

One fundamental difference between Yuasa and Dijkstra algorithms is that in the presence of a *Root Sets* transformation, installation collectors must never use the *Root Rescan elimination*, while deletion collectors have no requirement to apply it. The rescanning in deletion collectors is done mostly to eliminate floating garbage, albeit, at the expense of triggering work to rescan the roots.

Also, although at the end of our derivation, we arrived at a Yuasa algorithm, the result of every intermediate step is a valid deletion collector.

4.3 Derivation of a Hybrid Algorithm

The third derived practical algorithm is the Hybrid collector. The Hybrid algorithm is an instance of an abstract deletion collector. The Hybrid algorithm can be derived by applying the following transformations:

1. *Root Sets* transformation
2. *Shade compression* to tri-color
3. *SRC compression* to a single sticky bit
4. *Conflation of Shade and SRC*
5. *Root Rescan elimination for existing objects*
6. *Over approximate Shade*

The first two transformation steps are the same as for the Yuasa algorithm. However, in the Hybrid algorithm, we utilize the rescanning of roots only for newly allocated objects. The roots rescan transformation is parameterized to be active only for existing objects. The idea is to obtain a deletion Yuasa algorithm for the existing heap graph while maintaining a less restricted policy for newly allocated objects, similarly to the Dijkstra collector. By eliminating rescanning for existing objects, we can remove the *SRC* for those objects. Whenever the collector encounters an existing object during its barrier processing phase it will always mark the object, without applying any selection criteria.

After step 5 we still have a working deletion algorithm, but we would like to obtain more of the properties of Yuasa, namely, bounded re-tracing of newly allocated objects triggered by roots rescanning. To do that, we perform an additional transformation where if a newly allocated pointer is stored into the heap, the object is marked as reachable for this collection cycle. This simply means that if a newly allocated pointer is stored into the heap, we always increase the *SRC*, ignoring what the color of the destination object is. This is clearly a trivial over-approximation transformation on Shade of the destination object as indicated in step 6. With this, the collector now only needs to trace from existing objects and not from newly allocated objects. Newly allocated objects are essentially allocated white and colored black either during roots rescanning or during a pointer store in the heap.

The *Hybrid* collector is particularly suited for hard real-time applications, where it is desirable to achieve a bound on the roots rescanning work while reducing the floating garbage.

The skeleton code for the algorithm is illustrated in Fig. 5. $MarkRootsDirect$ is the procedure that performs the one-level deep rescanning procedure for the roots partition while the $isAllocatedInThisCycle$ bit is used to differentiate between newly allocated and existing objects.

5 Experimental Evaluation

We have implemented a concurrent collector framework in IBM's J9 virtual machine. The collector supports both standard work-based collection (for every a units of allocation the collector performs ka units of collection work) as well as time-based collection (the collector runs for c out of q time units). This collector has been built as a second-generation Metronome real-time collector [4].

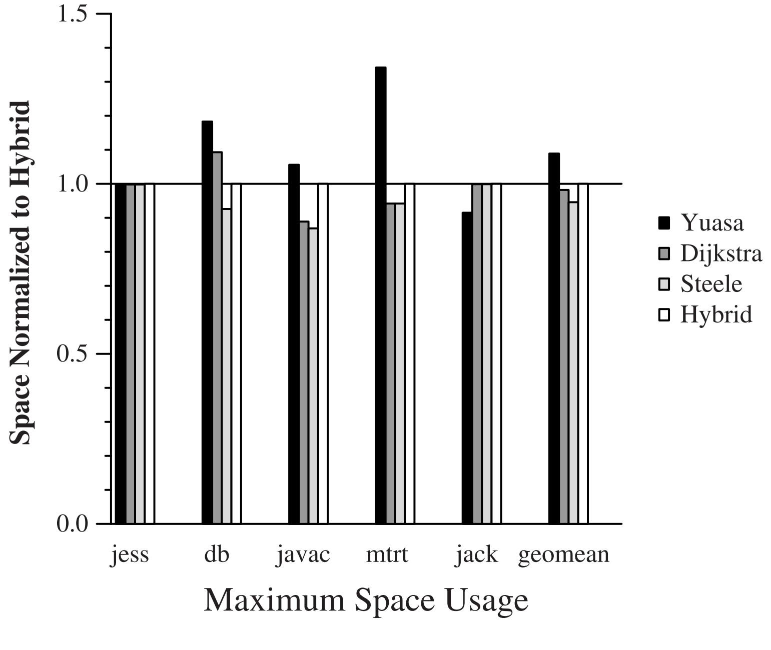


Fig. 6. Summary of the maximal space usage of the four collector algorithms. Data is normalized to the Hybrid algorithm. Shorter bars represent lower space usage

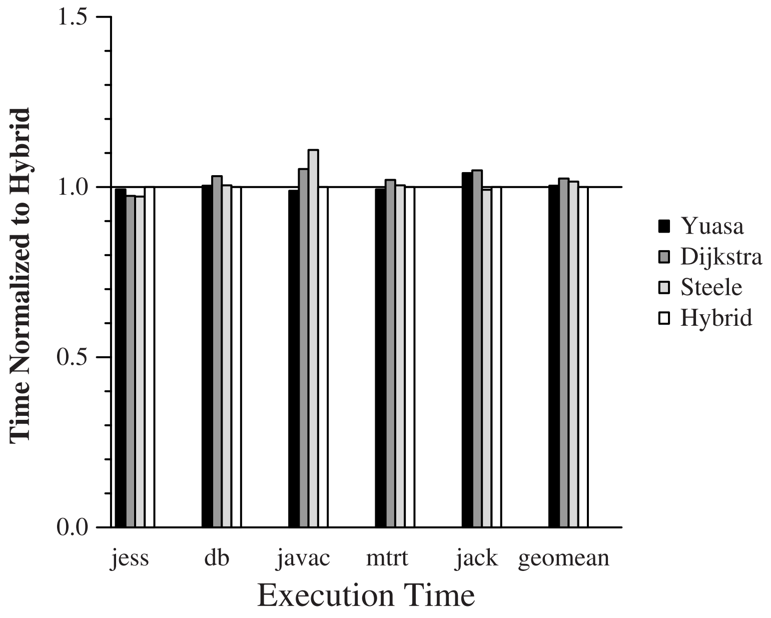


Fig. 7. Summary of overall execution time of the four collector algorithms. Data is normalized to the Hybrid algorithm. Shorter bars represent faster execution time

However, in this paper we will concentrate on work-based collection because its use is more common in more widely used soft real-time systems, and is likely to provide a better basis for comparison with other work. Isolated experiments have shown that the trends we report for work-based collection generally hold for time-based collection as well.

Our collector is implemented in a J2ME-based system that places a premium on space in the virtual machine. Therefore, we use the microJIT rather than the much more resource-intensive optimizing compiler. The microJIT is a high-quality single-pass compiler, producing code roughly a factor of 2 slower than the optimizing JIT.

The system runs on Linux/x86, Windows/x86, and Linux/ARM. The measurements presented here were performed on a Windows/x86 machine with a Pentium 4 3GHz CPU and 500MB of RAM.

The measurements presented all use a collector to mutator work ratio of 1.5, that is, for every 6K allocated by the mutator, the collector processes 9K. Collection is triggered when heap usage reaches 10MB.

We have measured the SPECjvm98 benchmarks, which exhibit a fairly wide range of allocation behavior (with the exception of compress, which performs very little allocation).

Figure 7 summarizes the performance of the four collector algorithms. The left graph shows the maximum heap size, the right graph total execution time. Both graphs are normalized to the Hybrid algorithm, and shorter bars represent better performance (less heap usage or shorter execution times). A geometric mean is also shown. These graphs summarize more detailed performance data which can be found in the Appendix.

5.1 Space Consumption

As expected, the incremental update collectors (Dijkstra and Steele) often require less memory than the snapshot collector (Yuasa). This is because the incremental update collectors allocate white (unmarked) and only consider live those objects which are added to the graph. However, there is no appreciable difference on 2 of the five benchmarks (jess and jack), which confirms that the space savings from incremental update collectors are quite program-dependent.

The use of Steele's write barrier instead of Dijkstra's theoretically produces less floating garbage at the expense of more re-scanning, since it marks the source rather than the target object of a pointer update. This means that if there are multiple updates to the same object, only the most recently installed pointer will be re-scanned.

However, the Steele barrier only leads to significant improvement in one of the benchmarks (db). This is because db spends much of its time performing sort operations. These operations permute the pointers in an array, and each update triggers a write barrier. With a Steele barrier, the array is tagged for re-scanning. But with a Dijkstra barrier, each object pointed to by the array is tagged for re-scanning. As a result, there is a great deal more floating garbage because the contents of the array are being changed over time.

Finally, the hybrid collector which we introduced, a snapshot collector that allocates white (unmarked), significantly reduces the space overhead of snapshot collection: the space overhead over the best collector is at worst 13% (for javac), which is quite reasonable.

5.2 Execution Time

While the incremental update collectors are generally assumed to have an advantage in space, their potential time cost is not well understood. Incremental update collectors may have to repeatedly re-scan portions of the heap that changed during tracing. Termination could be difficult if the heap is being mutated very quickly.

Our measurements show that incremental update collectors do indeed suffer time penalties for their tighter space bounds. The Dijkstra barrier causes significant slowdown in db, javac, mtrt, and jack. The Steele barrier is less prone to slowdown – only suffering on javac – but it does suffer the worst slowdown, about 12%. These measurements are total application run-time, so the slow-down of the collector is very large – this represents about a factor of 2 slowdown in collection time.

Once again, our hybrid collector performs very well – it usually takes time very close to the fastest algorithm. Thus the hybrid collector appears to be a very good compromise between snapshot and incremental update collectors.

Because its only rescanning is of the stack, it suffers no reduction in incrementality from a standard Yuasa-style collector, which must already scan the stack atomically. Its advantage over a standard snapshot collector is that it significantly reduces floating garbage by giving newly allocated objects time to die. But because it never rescans the heap, it avoids the termination problems of incremental update collectors and is still suitable for real-time applications.

As shown by the more detailed graphs in the appendix, the primary reason why the Yuasa and Hybrid algorithms are quicker is that the Dijkstra and Steele collectors both scan significantly more data during barrier buffer processing.

The benchmark with the most unusual behavior is jack, for which the Yuasa snapshot collector uses the *least* memory, while the Steele algorithm uses the least time. We are still in the process of investigating this behavior.

6 Conclusions

We have presented an abstract concurrent garbage collection algorithm and showed how incremental update collectors in the style of Dijkstra, and snapshot collectors in the style of Yuasa, can be derived from this abstract algorithm by reducing precision through various transformations.

We have also used the insights from this formulation to derive a new type of Hybrid snapshot collector which allocates its objects unmarked, and therefore induces less floating garbage.

We have implemented all four collectors in a production virtual machine and compared their time and space requirements. Incremental update collectors do indeed suffer less floating garbage, while the pure snapshot collector sometimes uses significantly more memory. The Hybrid collector greatly reduces the space cost of snapshot collection.

Incremental update collectors can significantly slow down garbage collection, leading to noticeable slow-downs in application execution speed. Our new Hybrid snapshot collector is generally about as fast as the fastest algorithm. For most applications, this collector will represent a good compromise between time and space efficiency, and has the notable advantage of snapshot collectors in terms of predictable termination.

We hope this work will spur further systematic study of algorithms for concurrent collection and further quantitative evaluation of those algorithms.

References

[1] Appel, A. W., Ellis, J. R., and Li, K. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, June 1988). *SIGPLAN Notices*, *23*, 7 (July), 11–20.

[2] Azatchi, H., Levanoni, Y., Paz, H., and Petrank, E. An on-the-fly mark and sweep garbage collector based on sliding views. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications* (Oct 2003), ACM Press, pp. 269–281.

[3] Bacon, D. F., Attanasio, C. R., Lee, H. B., Rajan, V. T., and Smith, S. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (Snowbird, Utah, June 2001). *SIGPLAN Notices*, *36*, 5 (May), 92–103.

[4] Bacon, D. F., Cheng, P., and Rajan, V. T. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, Jan. 2003). *SIGPLAN Notices*, *38*, 1, 285–298.

[5] Bacon, D. F., Cheng, P., and Rajan, V. T. A unified theory of garbage collection. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages, and Applications* (Vancouver, British Columbia, Oct. 2004), pp. 50–68.

[6] Baker, H. G. List processing in real-time on a serial computer. *Commun. ACM 21*, 4 (Apr. 1978), 280–294.
[7] Baker, H. G. The Treadmill, real-time garbage collection without motion sickness. *SIGPLAN Notices 27*, 3 (Mar. 1992), 66–70.
[8] Barth, J. M. Shifting garbage collection overhead to compile time. *Commun. ACM 20*, 7 (July 1977), 513–518.
[9] Ben-Ari, M. Algorithms for on-the-fly garbage collection. *ACM Trans. Program. Lang. Syst. 6*, 3 (1984), 333–344.
[10] Boehm, H.-J., Demers, A. J., and Shenker, S. Mostly parallel garbage collection. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation* (1991), ACM Press, pp. 157–164.
[11] Brooks, R. A. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming* (Austin, Texas, Aug. 1984), G. L. Steele, Ed., pp. 256–262.
[12] Cheadle, A. M., Field, A. J., Marlow, S., Peyton Jones, S. L., and While, R. L. Non-stop Haskell. In *Proc. of the Fifth International Conference on Functional Programming* (Montreal, Quebec, Sept. 2000). *SIGPLAN Notices*, *35*, 9, 257–267.
[13] Cheng, P., and Blelloch, G. E. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (Jun 2001), ACM Press, pp. 125–136.
[14] Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S., and Steffens, E. F. M. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM 21*, 11 (1978), 966–975.
[15] Domani, T., Kolodner, E. K., Lewis, E., Salant, E. E., Barabash, K., Lahan, I., Levanoni, Y., Petrank, E., and Yanorer, I. Implementing an on-the-fly garbage collector for java. In *Proceedings of the second international symposium on Memory management* (Oct 2000), ACM Press, pp. 155–166.
[16] Domani, T., Kolodner, E. K., and Petrank, E. A generational on-the-fly garbage collector for Java. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (June 2000). *SIGPLAN Notices*, *35*, 6, 274–284.
[17] Henriksson, R. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998.
[18] Hudson, R. L., and Moss, E. B. Incremental garbage collection for mature objects. In *Proc. of the International Workshop on Memory Management* (St. Malo, France, Sept. 1992), Y. Bekkers and J. Cohen, Eds., vol. 637 of *Lecture Notes in Computer Science*.
[19] Johnstone, M. S. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, University of Texas at Austin, Dec. 1997.
[20] Lamport, L. Garbage collection with multiple processes: an exercise in parallelism. In *Proc. of the 1976 International Conference on Parallel Processing* (1976), pp. 50–54.
[21] Larose, M., and Feeley, M. A compacting incremental collector and its performance in a production quality compiler. In ISMM [**?**], 1–9.
[22] Levanoni, Y., and Petrank, E. An on-the-fly reference counting garbage collector for java. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications* (Oct 2001), ACM Press, pp. 367–380.
[23] Nettles, S., and O'Toole, J. Real-time garbage collection. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (June 1993). *SIGPLAN Notices*, *28*, 6, 217–226.
[24] North, S. C., and Reppy, J. H. Concurrent garbage collection on stock hardware. In *Functional Programming Languages and Computer Architecture* (Portland, Oregon, Sept. 1987), G. Kahn, Ed., vol. 274 of *Lecture Notes in Computer Science*, pp. 113–133.

[25] Pixley, C. An incremental garbage collection algorithm for multi-mutator systems. *Distributed Computing 3, 1 6*, 3 (Dec. 1988), 41–49.
[26] Roth, D. J., and Wise, D. S. One-bit counts between unique and sticky. In ISMM [**?**], pp. 49–56.
[27] Steele, G. L. Multiprocessing compactifying garbage collection. *Commun. ACM 18*, 9 (Sept. 1975), 495–508.
[28] Steele, G. L. Corrigendum: Multiprocessing compactifying garbage collection. *Commun. ACM 19*, 6 (June 1976), 354.
[29] Yuasa, T. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software 11*, 3 (Mar. 1990), 181–198.

Appendix: Detailed Performance Data

This section includes graphs that illustrate for each benchmark, the behavior of the four collectors with respect to space utilization and barrier-induced work.

Figure 8 shows space usage over time by javac. Each data point represents the amount of data in use when the tracing and barrier processing terminated, but before sweeping. This represents the point of maximum memory use. The Yuasa-style collector consistently uses more memory than the others, but it also terminates the quickest (at termination, memory consumption is 0).

The reason why the Yuasa and Hybrid algorithms are quicker can easily be seen in Figure 9: the Dijkstra and Steele collectors both scan significantly more data during barrier buffer processing. Note that barrier-induced scanning is still significant even for the pure snapshot (Yuasa) collector. This is because pointers to some objects that are part of the snapshot may have been overwritten and not discovered during marking. Therefore, the snapshot it "completed" during barrier buffer processing. However, the total work will be based on the live data in the object graph at the time collection began, whereas in the incremental update algorithms it varies.

The rescanning overhead that we observed above for the db benchmark with Dijkstra's barrier can be seen clearly in Figure 11: rescanning typically causes about 20% of the heap to be re-visited, while rescanning for the other three collectors is negligible.

Details for the remaining benchmarks are found in Figures 12 through 17.

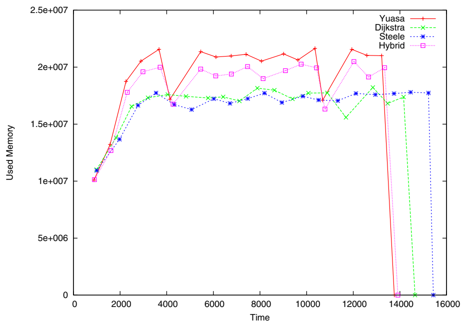


Fig. 8. Space vs. Time: javac

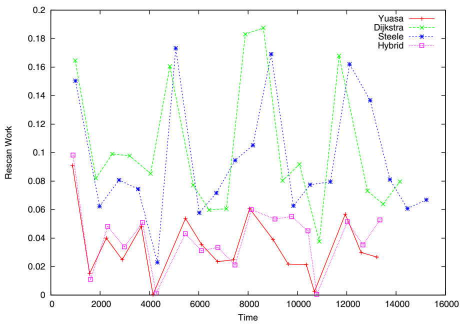


Fig. 9. Collector Rescanning Work: javac

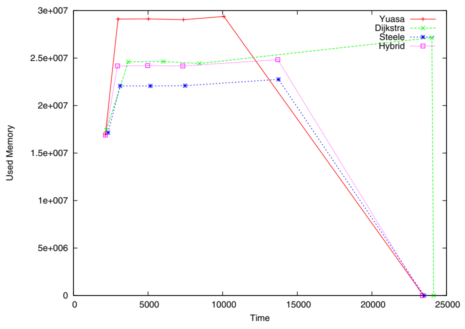


Fig. 10. Space vs. Time: db

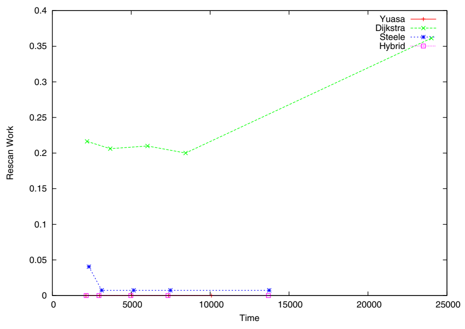


Fig. 11. Collector Rescanning Work: db

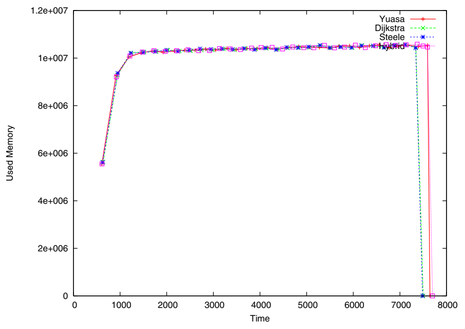


Fig. 12. Space vs. Time: jess

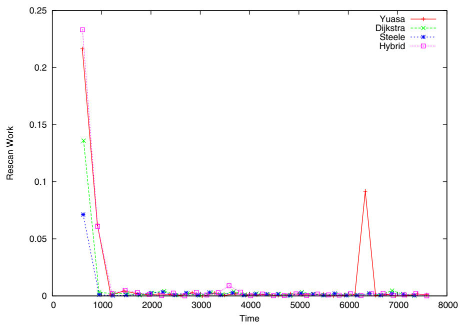


Fig. 13. Collector Rescanning Work: jess

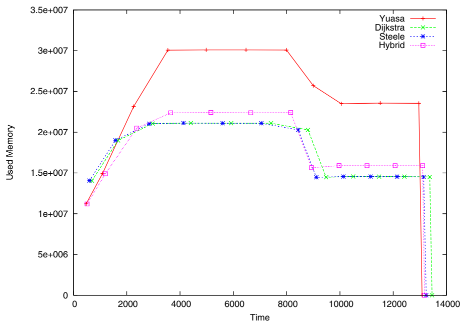


Fig. 14. Space vs. Time: mtrt

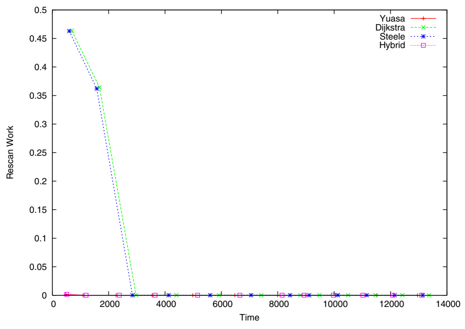


Fig. 15. Collector Rescanning Work: mtrt

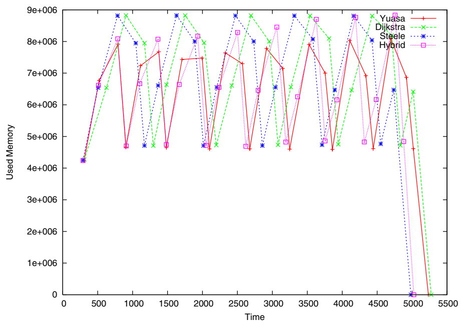


Fig. 16. Space vs. Time: jack

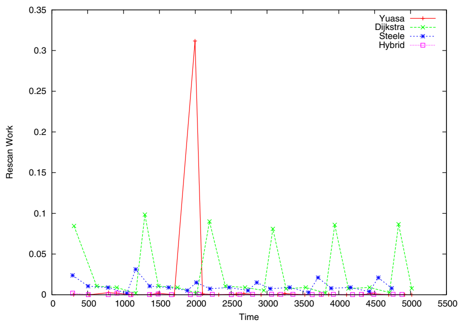


Fig. 17. Collector Rescanning Work: jack

Static Deadlock Detection for Java Libraries

Amy Williams, William Thies, and Michael D. Ernst

Computer Science and Artificial Intelligence Laboratory,
Massachusetts Institute of Technology,
Cambridge, MA 02139 USA
{amy, thies, mernst}@csail.mit.edu

Abstract. Library writers wish to provide a guarantee not only that each procedure in the library performs correctly in isolation, but also that the procedures perform correctly when run in conjunction. To this end, we propose a method for static detection of deadlock in Java libraries. Our goal is to determine whether client code exists that may deadlock a library, and, if so, to enable the library writer to discover the calling patterns that can lead to deadlock.

Our flow-sensitive, context-sensitive analysis determines possible deadlock configurations using a lock-order graph. This graph represents the order in which locks are acquired by the library. Cycles in the graph indicate deadlock possibilities, and our tool reports all such possibilities. We implemented our analysis and evaluated it on 18 libraries comprising 1245 kLOC. We verified 13 libraries to be free from deadlock, and found 14 distinct deadlocks in 3 libraries.

1 Introduction

Deadlock is a condition under which the progress of a program is halted as each thread in a set attempts to acquire a lock already held by another thread in the set. Because deadlock prevents an entire program from working, it is a serious problem.

Finding and fixing deadlock is difficult. Testing does not always expose deadlock because it is infeasible to test all possible interleavings of a program's threads. In addition, once deadlock is exhibited by a program, reproducing the deadlock scenario can be troublesome, thus making the source of the deadlock difficult to determine. One must know how the threads were interleaved to know which set of locks are in contention.

We propose a method for static deadlock detection in Java libraries. Our method determines whether it is possible to deadlock the library by calling some set of its public methods. If deadlock is possible, it provides the names of the methods and variables involved.

To our knowledge, the problem of detecting deadlock in libraries has not been investigated previously. This problem is important because library writers may wish to guarantee their library is deadlock-free for any calling pattern. For example, the specification for `java.lang.StringBuffer` in Sun's Java Development Kit (JDK) states:

```
class BeanContextSupport {
  protected HashMap children;

  public boolean remove(Object targetChild) {
    synchronized(BeanContext.
                 globalHierarchyLock) {
      ...
      synchronized(targetChild) {
        ...
        synchronized (children) {
          children.remove(targetChild);
        }
        ...
      }
    }
    return true;
  }

  public void
  propertyChange(PropertyChangeEvent pce) {
    ...
    Object source = pce.getSource();
    synchronized(children) {
      if ("beanContext".equals(propertyName)
          && containsKey(source)
          && ((BCSChild)children.get(source)).
             isRemovePending()) {
        BeanContext bc = getBeanContextPeer();
        if (bc.equals(pce.getOldValue())
            && !bc.equals(pce.getNewValue())) {
          remove(source);
        } else {
          ...
}}}}}
```

Fig. 1. Simplified code excerpt from the `BeanContextSupport` class in the `java.beans.beancontext` package of Sun's JDK

```
Object source
  = new Object();

BeanContextSupport support
  = new BeanContextSupport();

BeanContext oldValue
  = support.getBeanContextPeer();

Object newValue
 = new Object();

PropertyChangeEvent event
 = new PropertyChangeEvent(source,
                           "beanContext",
                           oldValue,
                           newValue);

support.add(source);
support.vetoableChange(event);

thread 1:  support.propertyChange(event);
thread 2:  support.remove(source);
```

Fig. 2. Client code that can cause deadlock in methods from Figure 1. In thread 1, `children` is locked, then `BeanContext.globalHierarchyLock` is locked (via a call to `remove`) while in thread 2, the ordering is reversed. Deadlock occurs under some thread interleavings. The initialization code shown above is designed to elicit the relevant path of control flow within the library

> The [`StringBuffer`] methods are synchronized where necessary so that all the operations on any particular instance behave as if they occur in some serial order that is consistent with the order of the method calls made by each of the individual threads involved.

If the operations are to behave as if they occurred in some serial order, deadlock between `StringBuffer` methods should not be possible. No serial ordering over the `StringBuffer` methods could lead to deadlock because locks acquired by Java's `synchronized` construct (which `StringBuffer` uses) cannot be held between method calls. Nonetheless, our tool reports a calling pattern that causes deadlock in `StringBuffer`.

Libraries are often vulnerable to deadlock. We have induced 14 distinct instances of deadlock in 3 libraries (for detailed results, see Section 6). Simplified code for one of the deadlocks found in Sun's JDK is shown in Figure 1. In the `BeanContextSupport` class of the `java.beans.beancontext` package, the `remove()` and `propertyChange()` methods obtain locks in a different order. The client code shown in Figure 2 can induce deadlock using these methods. Several other methods in the same package use the same locking order as `remove()` and thus exhibit the same deadlock vulnerability.

This deadlock has a simple solution: the `propertyChange()` method can synchronize on `BeanContext.globalHierarchyLock` before `children`, or it could lock only `globalHierarchyLock`. Section 6.1 describes solutions for other deadlocks.

An overview of our analysis is given in Section 3. We have implemented our technique and analyzed 18 libraries consisting of 1245k lines of code, obtained from SourceForge, Savannah, and other open source resources. Using our tool, we verified 13 of these libraries to be free of deadlock, and confirmed 14 distinct instances of deadlock in 3 libraries.

Detecting deadlock across all possible calls to a library is different than detecting deadlock in a whole program. Concrete aliasing relationships exist and can be determined for a whole program, whereas the analysis of a library must consider all possible calls into the library, which includes a large number of aliasing possibilities. In a program, the number of threads can often be determined, but a client may call into a library from any number of threads, so our analysis must model an unbounded number of threads. These differences combine to yield a much larger number of reports than would be present in a program, which makes it important to suppress false reports.

The remainder of this paper is organized as follows. Section 2 explains the semantics of locks in the Java programming language. Section 3 discusses our analysis at a high level, and Section 4 provides a more detailed description of the analysis. Section 5 describes techniques for reducing the number of spurious reports. Section 6 gives our experimental results. Related work is given in Section 7, and Section 8 concludes.

2 Locks in Java

In Java, each object conceptually has an associated lock; for brevity, we will sometimes speak of an object as being a lock. The Java "`synchronized (`*expr*`) {` *statements* `}`" statement evaluates the expression to an object reference, acquires the lock, evaluates the statements in the block, and releases the lock when the block is exited, whether normally or because of an exception. This design causes locks to be acquired in some order and then released in reverse (that is, in LIFO order), a fact that our analysis takes advantage of. A Java method can be declared `synchronized`, which is syntactic sugar for wrapping the body in `synchronized (this) { ... }` for instance methods, or `synchronized (`*C*`.class) { ... }`, where *C* is the class containing the method, for static methods.

A lock that is held by one thread cannot be acquired by another thread until the first one releases it. A thread blocks if it attempts to acquire a lock that is held by another thread, and does not continue processing until it successfully acquires the lock.

A lock is held per-thread; if a given thread attempts to re-acquire a lock, then the acquisition always succeeds without blocking.[1] The lock is released when exiting the `synchronized` statement that acquired it.

The `wait()`, `notify()`, and `notifyAll()` methods operate on receivers whose locks are held. An exception is thrown if the receiver's lock is not held. The `wait()` method releases the lock on the receiver object and places the calling thread in that object's wait set. While a thread is in an object's wait set, it is not scheduled for processing. Threads are reenabled for processing via the `notify()` and `notifyAll()` methods, which, respectively, remove one or all the threads from the receiver object's wait set. Once a thread is removed from an object's wait set, the `wait()` method attempts to reacquire the lock for the object it was invoked on. The `wait()` method returns only after the lock is reacquired. Thus, a thread may block inside `wait()` as it attempts to reacquire the lock for the receiver object.

Java 1.5 introduces new synchronization mechanisms in the `java.util.concurrent` package that allow a programmer to acquire and release locks without using the `synchronized` keyword. These mechanisms make it possible to acquire and release locks in any order (in particular, acquires and releases need not be in LIFO order). Our tool does not handle these new capabilities in the Java language. However, most synchronization can be expressed using the primitives from Java 1.4, and we therefore expect that our technique will be applicable under current and future releases of Java.

3 Analysis Synopsis

We consider a *deadlock* to be the condition in which a set of threads cannot make progress because each is attempting to acquire a lock that is held by another member of the set. Our deadlock detector uses an interprocedural analysis to track possible sequences of lock acquisitions within a Java library. It represents possible locking patterns using a graph structure—the *lock-order graph*, described below. Cycles in this graph indicate possibilities of deadlock.

For each cycle, our tool reports the variable names of the locks involved in the deadlock as well as the methods that acquire those locks (see Section 4.4). Our tool is conservative and reports all deadlock possibilities. However, the conservative approximations cause the tool to consider infeasible paths and impossible alias relationships, resulting in false positives (spurious reports).

3.1 Lock-Order Graph

The analysis builds a single lock-order graph that captures locking information for an entire library. This graph represents the order in which locks are acquired

[1] For our purposes, it is sufficient to consider multiple synchronized statements over the same object in one thread as a no-op. A Java virtual machine tracks the number of *lock*/*unlock* actions (entrance and exit of a `synchronized` block) for each object. A counter is updated for each `synchronized` statement, but if the current thread already holds the target lock, no change is made to the thread's lock set.

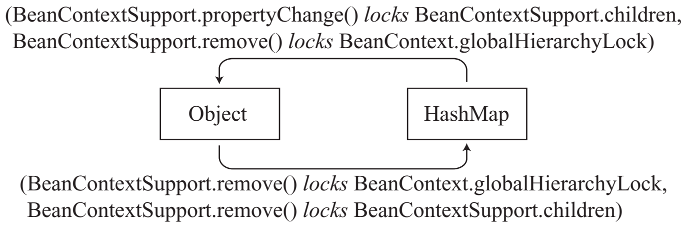

(BeanContextSupport.propertyChange() *locks* BeanContextSupport.children,
BeanContextSupport.remove() *locks* BeanContext.globalHierarchyLock)
Object
HashMap
(BeanContextSupport.remove() *locks* BeanContext.globalHierarchyLock,
BeanContextSupport.remove() *locks* BeanContextSupport.children)

Fig. 3. Relevant portion of the lock-order graph for the code in Figure 1. The nodes represent the set of all `Object`s and `HashMap`s, respectively. Each edge is annotated by the sequence of methods (and corresponding variable names) that acquire first a lock from the source set, then a lock from the destination set

via calls to the library's public methods. Combining information about the locking behavior of each public method into one graph allows us to represent any calling pattern of these methods across any number of threads.

Each node of the lock-order graph represents a set of objects that may be aliased. (Types are an approximation to may-alias information; Section 5.1 gives a finer but still lightweight approximation applicable to fields.) An edge in the graph indicates nested locking of objects along some code path. That is, it indicates the possibility of locking first an object from the source node, then an object from the destination node.

A cycle consisting of nodes N_1 and N_2 means that along some code path, an object $o_1 \in N_1$ may be locked before some object $o_2 \in N_2$, and along another (or the same) path, o_2 may be locked before o_1. In general, a cycle exposes code paths leading to cyclic lock orders, and, when the corresponding paths are run in separate threads, deadlock may occur. Figure 3 shows the lock-order graph for the code in Figure 1.

To build the graph, the analysis iterates over the methods in the library, building a lock-order graph for each of them. All possible locking configurations of a method are modeled, including locks acquired transitively via calls to other methods. At a call site, the callee's graph is inserted into the caller. After each method's lock-order graph has reached a fixed point, the public methods' lock-order graphs are merged into a single graph for the library. Cycles are then detected, and reports are generated.

3.2 Deadlocks Detected by Our Technique

Our goal is to detect cases in which a sequence of client calls can cause deadlock in a library, or to verify that no such sequence exists. Our tool reports deadlock possibilities in which all deadlocked threads are blocked within a single library, attempting to acquire locks via Java `synchronized` statements or `wait()` calls. Under certain assumptions about the client and the library, our tool reports all such possibilities.

Our analysis focuses on deadlocks due to lock acquisitions via Java `synchronized` statements and `wait()` calls: progress of a program is halted as each thread in a set attempts to acquire a lock already held by another thread in the set. We are not concerned with other ways in which a program may fail to make progress. A thread might hang forever while waiting for input, enter an infinite loop, suffer livelock, or fail to call `notify()` or to release a user- or library-defined lock (that is, using a locking mechanism not built into Java). These problems in one thread can prevent another thread or the whole program from making progress: consider a call to `Thread.join()` (which waits for a given thread to terminate) on a thread that does not terminate. Detecting all of these problems is outside the scope of this paper.

Assumptions About Client Code. We make three assumptions about client code. If a client deviates from these assumptions, our tool is still useful for detecting deadlock, but it cannot detect deadlocks introduced by the deviant behavior. First, we assume that the client does not include a class that extends a library class or belongs to a library package. If such a class exists, it needs to be inspected by our analysis and treated as part of the library. Second, we assume that that the client does not invoke library methods within callbacks from the library; that is, all client methods M are either unreachable from the library, or the library is unreachable from M. For example, if a client class overrides `Object.hashCode()` such that it calls a synchronized method in the library, then any library method calling `hashCode()` should model that synchronization. The class therefore needs to be analyzed as though it is part of the library. Third, we assume that the client code is *well-behaved*: either it does not lock any objects locked by the library, or it does so in disciplined ways (as explained below).

Without the assumption of well-behavedness, it is difficult or impossible to guarantee deadlock freedom for a library without examining client code. An adversarial client can induce deadlock if it has access to two objects locked by a library. For example, suppose that a library has a synchronized method:

```
class A {
  synchronized void foo() { ... }
}
```

Then a client could cause deadlock in the following way:

```
A a1 = new A(), a2 = new A();
thread 1:  synchronized(a1) { a2.foo(); }
thread 2:  synchronized(a2) { a1.foo(); }
```

A client that locks a different set of objects than those locked by the library is always well-behaved. This is the case for arbitrary clients if the locks used by the library do not escape it; that is, if they are inaccessible to the client. Section 5.1 describes a method for detecting some inaccessible locks.

Even if the client and the library share a set of locks, the client can be well-behaved if it acquires those locks in a restricted pattern. These restrictions could be part of the library's specification—and such documentation could even be automatically generated for the library by a tool like ours. As above, one sufficient restriction is that clients do not lock objects that the library may lock; this requires the library to specify the set of objects that it will lock. A more liberal but sufficient restriction is that the client acquires locks in an order compatible with the library. In this scenario, the library specifies the order of lock acquisitions (say, as a lock-order graph), and clients are forbidden from acquiring locks in an order that introduces cycles into the graph. We believe that these restrictions are quite reasonable, and that information about the locks acquired by a library are a desirable part of its specification.

Assumptions About Library Code. In practice, libraries do not exist in isolation. Rather, each library uses additional libraries (e.g., the JDK) to help it accomplish its task. One approach to analyzing such cascaded libraries is to consider all of the libraries together, as if they were a single library. However, this hampers modularity, as the guarantees offered for one library depend on the implementation of other libraries. It also hampers scalability, as the effective library size can grow unwieldy for the analysis. For these reasons, our analysis considers each library independently. Consider that the "main" library under consideration relies on several "auxiliary" libraries. Under certain assumptions about the main library, our analysis detects all deadlock possibilities in which all threads are blocked within the main library. It does not report cases in which some threads are blocked in the main library and other threads are blocked in auxiliary libraries.

We make the following assumptions about library code. First, as the library under consideration (the main library) may be a client of some auxiliary libraries, it must satisfy the client assumptions (described previously) to guarantee deadlock freedom for its own users. Second, the main library cannot perform any synchronization in methods that are reachable via callbacks from auxiliary libraries (e.g., in `Object.hashCode()`). Callbacks through the auxiliary libraries are inaccessible to the analysis. Third, the library cannot use reflection. Reflection can introduce opaque calling sequences that impact the lock ordering. As with the client code, our analysis operates as usual even if these assumptions are broken, but it can no longer guarantee that all deadlock possibilities are reported.

4 Algorithm Details

The deadlock detector employs an interprocedural dataflow analysis for constructing lock-order graphs. The analysis is flow-sensitive and context-sensitive. At each program point, the analysis computes a symbolic state modeling the library's execution state. The symbolic state at the end of a method serves as a method summary. The analysis is run repeatedly over all methods until a fixed point is reached; termination of the analysis is guaranteed.

The type domains for the analysis are given in Figure 4. For simplicity, we present the algorithm for a language that models the subset of Java relevant to our analysis. The language omits field assignments; they are not relevant because

$$
\begin{array}{rl}
T \in & \text{Type} \\
v \in & \text{LocalVar} \\
\mathit{method} \in & \text{MethodDecl} = T_r\ m(T_1\ v_1,\ T_2\ v_2,\ \ldots,\ T_n\ v_n)\ \{\ \mathit{stmt}\ \} \\
& \qquad\qquad\qquad \text{where } v_1 = \texttt{this} \text{ if } m \text{ is instance method} \\
\mathit{library} \in & \text{Library} = \text{set-of MethodDecls} \\
\mathit{stmt} \in & \text{Statement} = \begin{array}{ll} \ \ T\ v & \mid \texttt{branch}\ \mathit{stmt}_1\ \mathit{stmt}_2 \\ \mid v := \texttt{new}\ T & \mid \texttt{synchronized}\ (v)\ \{\ \mathit{stmt}\ \} \\ \mid v_1 := v_2 & \mid v := m(v_1, \ldots, v_n) \\ \mid v_1 := v_2.f & \mid \texttt{wait}(v) \\ \mid \mathit{stmt}_1;\ \mathit{stmt}_2 & \end{array} \\
\mathit{pp} \in & \text{ProgramPoint}_\perp \\
o = \langle \text{pp}, \text{T} \rangle \in & \text{HeapObject} = \text{ProgramPoint} \times \text{Type} \\
g \in & \text{Graph} = \text{directed-graph-of HeapObjects} \\
\mathit{roots} \in & \text{Roots} = \text{set-of HeapObjects} \\
\mathit{env} \in & \text{Environment} = \text{LocalVar} \rightarrow \text{HeapObject} \\
s = \langle \text{g}, \text{roots}, \text{locks}, \in & \text{State} = \text{Graph} \times \text{Roots} \times \text{list-of HeapObjects} \times \\
\text{env}, \text{wait} \rangle \quad & \qquad\qquad \text{Environment} \times \text{set-of HeapObjects}
\end{array}
$$

Fig. 4. Type domains for the lock-order dataflow analysis. Parameters are considered to be created at unique points before the beginning of a method. The "`branch` *stmt*$_1$ *stmt*$_2$" statement is a non-deterministic branch to either *stmt*$_1$ or *stmt*$_2$

our analysis does not track the flow of values through fields. Synchronized methods are modeled in this language using their desugaring (see Section 2) and loops are supported via recursion. Our implementation handles the full Java language.

Our analysis operates on symbolic heap objects. Each symbolic heap object represents the set of objects created at a given program point [6]; it also contains their type. For convenience, we say that a symbolic heap object o is *locked* when a particular concrete object drawn from o is locked.

The **state** is a 5-tuple consisting of:

- The current lock-order **graph**. Each node in the graph is a symbolic heap object. The graph represents possible locking behavior for concrete heap objects drawn from the sets modeled by the symbolic heap objects. A path of nodes $o_1 \ldots o_k$ in the graph corresponds to a potential program path in which o_1 is locked, then o_2 is locked (before o_1 is released), and so on.
- The **roots** of the graph. The roots represent objects that are locked at some point during execution of a given method when no other lock is held.
- The list of **locks** that are currently held, in the order in which they were obtained.
- An **environment** mapping local variables to symbolic heap objects. The environment is an important component of the interprocedural analysis, as it allows information to propagate between callers and callees. It also improves precision by tracking the flow of values between local variables.
- A set of objects that have had **wait** called on them without an enclosing `synchronized` statement in the current method.

4.1 Dataflow Rules

The dataflow rules for the analysis are presented in Figure 5. Helper functions appear in Figure 6, and mathematical operators (including the join operator) are defined in Figure 7. Throughout the following explanation, we define the *current lock* as the most recently locked object whose lock remains held; it is the last object in the list of currently held locks, or tail(s.locks).

The symbolic state is updated in the **visit_stmt** procedure (in Figure 5) which visits each statement in a method. A variable declaration or initialization introduces a fresh heap object. An assignment between locals copies an object within the local environment. A field reference introduces a fresh object (the analysis does not model the flow of values through fields). A `branch` models divergent paths and is handled by the join operator below. Calls to `wait()` are described in Section 4.2.

The rule for `synchronized` statements handles lock acquires; there are two cases. First, if the target object o is not currently locked (i.e., if $o \notin s$.locks), then an edge is added to the lock-order graph from the current lock to o, and o is appended to s.locks. If no objects were locked before the `synchronized` statement, o becomes a root in the graph (roots are important at a call site, as discussed below). Next, the analysis descends into the body of the `synchronized` block. Upon completion, the analysis continues to the next statement, preserving the lock-order graph from the `synchronized` block but restoring the list of locked objects valid before the `synchronized` statement. This is correct, since Java's syntax guarantees that any objects locked within the `synchronized` block are also released within the block.

In the second case for `synchronized` statements, the target is currently locked. Though the body is analyzed as before, the synchronization is a no-op and does not warrant an edge in the lock-order graph. To exploit this fact, the analysis needs to determine whether nested `synchronized` statements are locking the same concrete object. Though symbolic heap objects represent *sets* of concrete objects, they nonetheless can be used for this determination: if nested `synchronized` statements lock variables that are mapped to the same heap object (during analysis), then they always lock the same concrete object (during execution). This is true within a method because each heap object is associated with a single program point; as this simplified language contains no loops, any execution will visit that point at most once and hence create at most one concrete instance of the heap object. This notion also extends across methods, as both heap objects and concrete objects are directly mapped from caller arguments into callee parameters as described below. Thus, repeated synchronization on a given heap object is safely ignored, significantly improving the precision of the analysis.

Method calls are handled by integrating the graph for the callee into the caller as follows. In the case of overridden methods, each candidate implementation's graph is integrated. The analysis uses the most recent lock-order graph that has been calculated for the callee. Recursive sequences are iterated until reaching a fixed point. The calling context is first incorporated into a copy of the callee's graph either by removing the formal parameters (if the corresponding argument

```
**visit_stmt**($stmt, s$) returns State $s'$
  $s' \leftarrow s$
  switch($stmt$)
    **case** $T\ v$ | $v$ := `new` $T$
      $s'$.env ← $s$.env[$v$ := ⟨ **program_point**($stmt$), $T$ ⟩]
    **case** $v_1 := v_2$
      $s'$.env ← $s$.env[$v_1$ := $s$.env[$v_2$]]
    **case** $v_1 := v_2.f$
      $s'$.env ← $s$.env[$v_1$ := ⟨ **program_point**($stmt$), declared_type($v_2.f$) ⟩]
    **case** $stmt_1$; $stmt_2$
      $s_1$ ← **visit_stmt**($stmt_1$, $s$)
      $s'$ ← **visit_stmt**($stmt_2$, $s_1$)
    **case** `branch` $stmt_1$ $stmt_2$
      $s'$ ← **visit_stmt**($stmt_1$, $s$) $\sqcup$ **visit_stmt**($stmt_2$, $s$)
    **case** `synchronized` ($v$) { $stmt$ }
      $o \leftarrow s$.env[$v$]
      if $o \in s$.locks then
        // already locked $o$, so synchronized statement is a no-op
        $s_1 \leftarrow s$
      else
        // add $o$ to $g$ under current lock, or as root if no locks held
        if $s$.locks is empty  // below, • denotes list concatenation
          then $s_1$ ← ⟨$s$.g ∪ $o$, $s$.roots ∪ $o$, $s$.locks • $o$, $s$.env, $s$.wait⟩
          else $s_1$ ← ⟨$s$.g ∪ $o$ ∪ edge(tail($s$.locks) → $o$), $s$.roots, $s$.locks • $o$,
                   $s$.env, $s$.wait⟩
      $s_2$ ← **visit_stmt**($stmt$, $s_1$)
      $s'$ ← ⟨$s_2$.g, $s_2$.roots, $s$.locks, $s_2$.env, $s_2$.wait⟩
    **case** $v := m(v_1, \ldots, v_n)$
      $s'$.env ← $s$.env[$v$ := ⟨ **program_point**($stmt$), return_type($m$) ⟩]
      ∀ versions of $m$ in subclasses of env[$v_1$].T:
        $s_m$ ← **visit_method**(method_decl($m$))
        $s'_m$ ← **rename_from_callee_to_caller_context**($s_m$, $s$, $n$)
        // connect the two graphs, including roots
        $s'$.g ← $s'$.g ∪ $s'_m$.g
        if $s$.locks is empty then  // connect current lock to roots of $s'_m$
          $s'$.roots ← $s'$.roots ∪ $s'_m$.roots
          $s'$.wait ← $s'$.wait ∪ $s'_m$.wait
        else
          ∀ $root \in s'_m$.roots:
            $s'$.g ← $s'$.g ∪ edge(tail($s$.locks) → $root$)
          ∀ $o \in s'_m$.wait:  if tail($s$.locks) ≠ $o$ then
            $s'$.g ← $s$.g ∪ $o$ ∪ edge(tail($s$.locks) → $o$)
    **case** `wait`($v$)
      $o \leftarrow s$.env[$v$]
      if $s$.locks is empty then
        $s'$.wait ← $s$.wait ∪ $o$
      else if tail($s$.locks) ≠ $o$ then
        // `wait` releases then reacquires $o$: new lock ordering
        $s'$.g ← $s$.g ∪ $o$ ∪ edge(tail($s$.locks) → $o$)
```

Fig. 5. Dataflow rules for the lock-order data-flow analysis

```
**program_point**($stmt$) returns the program point for statement $stmt$
**visit_method**($T_r\ m(T_1\ v_1, \ldots, T_n\ v_n)\ \{\ stmt\ \}$) returns State $s'$
  $s' \leftarrow$ empty State
  $\forall$ parameters $T_i\ v_i$ (including this):
    $s' \leftarrow$ **visit_stmt**($T_i\ v_i, s'$) // process formals via "$T\ v$" rule
  $s' \leftarrow$ **visit_stmt**($stmt, s'$)

**rename_from_callee_to_caller_context**($s_m, s, n$) returns State $s'_m$
  $s'_m \leftarrow s_m$
  $\forall j \in [1,n] : formal_j \leftarrow s_m.\mathrm{env}[v_j]$    // formal parameter
  $\forall j \in [1,n] : actual_j \leftarrow s.\mathrm{env}[v_j]$     // actual argument
  $\forall o \in s_m.\mathrm{g}$ :       // for all objects $o$ locked by the callee
    if $\exists\ j\ \ s.t.\ \ o = formal_j$
      // $o$ is formal parameter $j$ of callee method
      then if $actual_j \in s.\mathrm{locks}$
        // caller locked $o$, remove $o$ from callee graph
        then $s'_m.\mathrm{g}, s'_m.\mathrm{roots} \leftarrow$ **splice_out_node**($s_m.\mathrm{g}, s_m.\mathrm{roots}, o$)
        // caller did not lock $o$, rename $o$ to actual arg
        else $s'_m.\mathrm{g}, s'_m.\mathrm{roots} \leftarrow$ **replace_node**($s_m.\mathrm{g}, s_m.\mathrm{roots}, o, actual_j$)
      // $o$ is not from caller, rename $o$ to bottom program point $pp_\perp$
      else $s'_m.\mathrm{g}, s'_m.\mathrm{roots} \leftarrow$ **replace_node**($s_m.\mathrm{g}, s_m.\mathrm{roots}, o, \langle pp_\perp, o.\mathrm{T}\rangle$)
  $s'_m.\mathrm{wait} \leftarrow \emptyset$
  $\forall o \in s_m.\mathrm{wait}$       // for all objects in wait set
    if $\exists\ j\ \ s.t.\ \ o = formal_j$
      then $s'_m.\mathrm{wait} \leftarrow s'_m.\mathrm{wait}\ \cup\ actual_j$
      else $s'_m.\mathrm{wait} \leftarrow s'_m.\mathrm{wait}\ \cup\ \langle pp_\perp, o.\mathrm{T}\rangle$

**splice_out_node**($g, roots, o$) returns Graph $g'$, Roots $roots'$
  $g' \leftarrow g \setminus o$
  $\forall$ edges($src \rightarrow o$) $\in g\ \ s.t.\ \ o \neq src$ :
    $\forall$ edges($o \rightarrow dst$) $\in g\ \ s.t.\ \ o \neq dst$ :
      $g' \leftarrow g'\ \cup$ edge($src \rightarrow dst$)
  $roots' \leftarrow roots \setminus o$
  if $o \in roots$ then
    $\forall$ edges($o \rightarrow dst$) $\in g\ \ s.t.\ \ o \neq dst$ :
      $roots' \leftarrow roots' \cup dst$

**replace_node**($g, roots, o_{old}, o_{new}$) returns Graph $g'$, Roots $roots'$
  $g' \leftarrow (g \setminus o_{old})\ \cup\ o_{new}$
  $\forall$ edges($src \rightarrow o_{old}$) $\in g : g' \leftarrow g'\ \cup$ edge($src \rightarrow o_{new}$)
  $\forall$ edges($o_{old} \rightarrow dst$) $\in g : g' \leftarrow g'\ \cup$ edge($o_{new} \rightarrow dst$)
  if $o_{old} \in roots$
    then $roots' \leftarrow (roots \setminus o_{old})\ \cup\ o_{new}$
    else $roots' \leftarrow roots$
```

Fig. 6. Helper functions for the lock-order dataflow analysis

$actual_j$ is locked at the call site, in which case the lock acquire is a no-op from the caller's perspective) or by replacing them with the caller's actual arguments (if $actual_j$ is not locked at the call site). The non-formal parameter nodes are

```
$g_1 \cup g_2$ returns Graph $g'$
   // nodes are HeapObjects: equivalent values are collapsed
   $\text{nodes}(g') = \text{nodes}(g_1) \cup \text{nodes}(g_2)$
   // edges are pairs of HeapObjects: equivalent pairs are collapsed
   $\text{edges}(g') = \text{edges}(g_1) \cup \text{edges}(g_2)$

$g \setminus o$ returns Graph $g'$
   $\text{nodes}(g') = \text{nodes}(g) \setminus o$
   $\text{edges}(g') = \text{edges}(src \rightarrow dst) \in g \;\; s.t. \;\; o \neq src \;\wedge\; o \neq dst$

$s_1 \sqcup s_2$ returns State $s'$
   $s'.\text{g} \leftarrow s_1.\text{g} \cup s_2.\text{g}$
   $s'.\text{roots} \leftarrow s_1.\text{roots} \cup s_2.\text{roots}$
   $s'.\text{locks} \leftarrow s_1.\text{locks}$        // $s_1.\text{locks} = s_2.\text{locks}$
   $\forall v \in \{v' \mid v' \in s_1.\text{env} \;\vee\; v' \in s_2.\text{env}\}$ :
      if $s_1.\text{env}[v] = s_2.\text{env}[v]$
         then $s'.\text{env} \leftarrow s'.\text{env}[v := s_1.\text{env}(v)]$
         else $s'.\text{env} \leftarrow s'.\text{env}[v := \langle \textbf{program\_point}(\text{join\_point}(v)),\, T_1 \sqcup T_2 \,\rangle]$
   $s'.\text{wait} \leftarrow s_1.\text{wait} \cup s_2.\text{wait}$
$T_1 \sqcup T_2$ returns lowest common superclass of $T_1$ and $T_2$
```

Fig. 7. Union and difference operators for graphs, and join operator for symbolic state

then replaced with nodes of the same type and with a special program point of $pp_\perp$, indicating that they originated at an unknown program point (bottom). The callee's wait set is adjusted in a similar fashion. At this point, an edge is added from the current lock in the caller to each of the roots of the modified callee graph. Finally, the two graphs are merged, collapsing identical nodes and edges.

The **join** operator ($\sqcup$) in Figure 7 is used to combine states along confluent paths of the program (e.g., `if` statements). We are interested in locking patterns along any possible path, which, for the graphs, roots, and wait sets, is simply the union of the two incoming states' values. The list of current locks does not need to be reconciled between two paths, as the hierarchy of `synchronized` blocks in Java guarantees that both incoming states will be the same. The new environment remains the same for mappings common to both paths. If the mappings differ for a given variable then a fresh heap object must be introduced for that variable. The fresh object is assigned a program point corresponding to the join point for the variable (each variable is considered to join at a separate location). The strongest type constraint for the fresh object is the join of the variables' types along each path—their lowest common superclass.

The algorithm for constructing the entire library's lock-order graph is given in Figure 8. The **top_level** procedure first computes a fixed point state value for each method in the library. Termination is guaranteed since there can be at most $|\text{PP}| \cdot |\text{Type}|$ heap objects in a method and the analysis only adds objects to the graph at a given stage. After computing the fixed points, the procedure performs a post-processing step to account for subclassing. Because the analysis

```
**top_level**(*library*) returns Graph $g$
    $s_1, \ldots, s_n \leftarrow$ dataflow fixed points over public methods in *library*
    $g \leftarrow$ **post_process**$(s_1, \ldots, s_n)$

**post_process**$(s_1, \ldots, s_n)$ returns Graph $g$
    $g \leftarrow$ empty Graph
    $\forall i \in [1, n]$ :
        $\forall$ edges $(o_1 \rightarrow o_2) \in s_i$.g:
            // Add edges between all possible subclasses of locked objects.
            // All heap objects now have bottom program point $pp_\perp$.
            $\forall$ subclasses $T_1$ of $o_1.T$, $\forall$ subclasses $T_2$ of $o_2.T$:
                $o_{T_1} \leftarrow \langle pp_\perp, T_1 \rangle$
                $o_{T_2} \leftarrow \langle pp_\perp, T_2 \rangle$
                $g \leftarrow g \cup o_{T_1} \cup o_{T_2} \cup \text{edge}(o_{T_1} \rightarrow o_{T_2})$
```

Fig. 8. Top-level routine for constructing a lock-order graph for a library of methods

for each method was based on the declared type of locks, extra edges must be added for all possible concrete types that a given heap object could assume. While it is also possible to modify the dataflow analysis to deal with subclassing at each step, it is simpler and more efficient to use post-processing.

4.2 Calls to `wait()`

A call to `wait()` on object o causes the lock on o to be released and subsequently reacquired, which is modeled by adding an edge in the lock-order graph from the most recently acquired lock to o. However, this edge can be omitted if o is also the most recently acquired lock, as releasing and reacquiring this lock has no effect on the lock ordering. In contrast to `synchronized` statements, `wait()` can influence the lock-order graph even though its receiver is locked at the time of the call. For example, before the `wait()` call in Figure 9, `a` is locked before `b`. However, during the call to `wait()`, `a`'s lock is released and later acquired while `b`'s lock remains held, so `a` is also locked after `b`. Deadlock is therefore possible.

It is illegal to call `wait()` on an object whose lock is not held; if this happens during program execution, Java throws a runtime exception. Even so, it is possible for a method to call `wait()` outside any `synchronized` statement, since the receiver could be locked in the caller. When a method calls `wait()` outside any `synchronized` statement, our analysis needs to consider the calling context to determine the effects of the `wait()` call on the lock-order graph. For this reason, when no locks are held and `wait()` is called, the receiver object is stored in the wait set and later accounted for in a caller method.

None of the libraries we analyzed reported any potential deadlocks due to `wait()`. This suggests that programmers most often call `wait()` on the most recently acquired lock.

```
void m1(Object a, Object b) {      void m2(Object a, Object b) {      Object a = new Object();
  synchronized(a) {                  synchronized(a) {                Object b = new Object();
    synchronized (b) {                 a.notify();
      a.wait();                        synchronized (b) {             thread 1:  m1(a, b);
      ...                                ...                          thread 2:  m2(a, b);
}}}                                }}}
```

Fig. 9. Method `m1()` imposes both lock orderings `a`$\rightarrow$`b` and `b`$\rightarrow$`a`, due to the call to `a.wait()`. Method `m2()`, which imposes the lock ordering `a`$\rightarrow$`b`, can cause deadlock when run in parallel with `m1()`, as illustrated in the third column

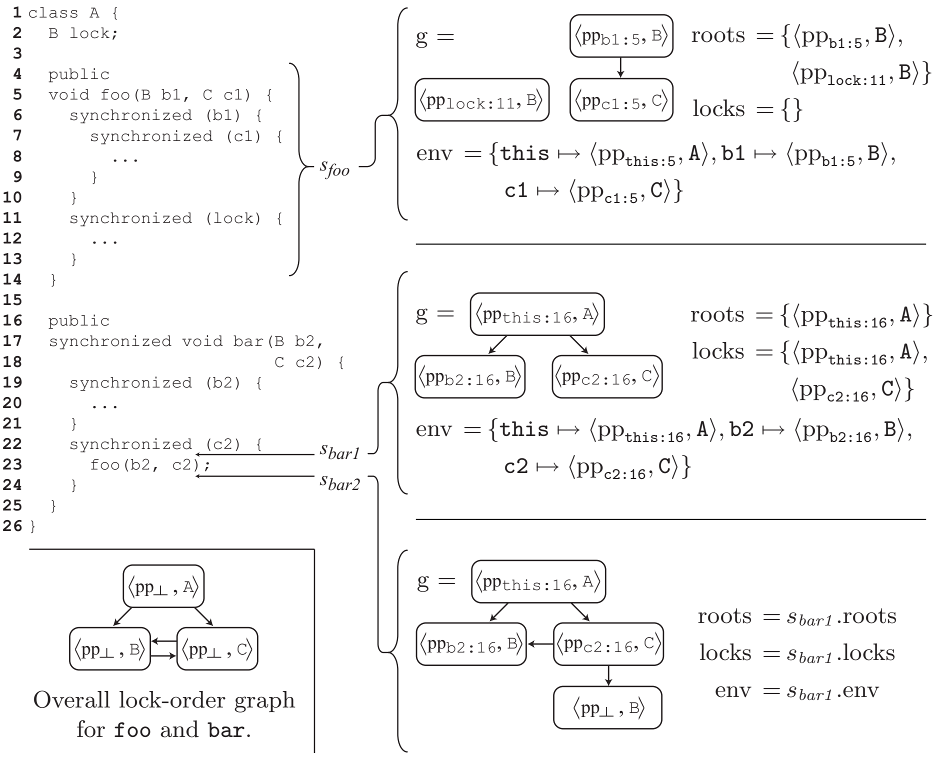
4.3 Dataflow Example

An example of the dataflow analysis appears in Figure 10. The example contains a class `A` with two methods, `foo()` and `bar()`. The symbolic state s_{foo} represents the method summary for `foo()`. Program points are represented as a variable name and a line number corresponding to the variable's assignment. For example, $\langle \mathrm{pp}_{\texttt{b1:5}}, \texttt{B} \rangle$ is a symbolic heap object, of type `B`, for parameter `b1` on line 5 of `foo()`; $\langle \mathrm{pp}_{\texttt{lock:11}}, \texttt{B} \rangle$ is a symbolic heap object, also of type `B`, for the field `lock` as referenced on line 11 of `foo()` (though `lock` is declared on line 2, each field reference creates a fresh heap object). The lock-order graph for `foo()` illustrates that parameters `b1` and `c1` can each be locked in sequence, with `lock` locked separately. Note that the graph contains two separate nodes for `b1` and `lock`—both of type `B`—in case one of them can be pruned when integrating into the graph of a caller.

The symbolic state in `bar()` immediately before the call to `foo()` is represented by s_{bar1}. Since `bar()` is a synchronized method, a heap object for `this` appears as a root of the graph. The graph illustrates that parameters `b2` and `c2` can be locked while the lock for `this` is held. The list of locks held at the point of the call is given by s_{bar1}.locks; it contains `this` and `c2`.

The most interesting aspect of the example is the method call from `bar()` to `foo()`. This causes the graph of s_{foo} to be adjusted for the calling context and then integrated into the graph of s_{bar1} with edges added from the node for the current lock, `c2`. The calling context begins with the actual parameter `b2`. Since `b2` is not locked in s_{bar1} at the point of the call, the formal parameter `b1` is replaced by `b2` throughout the graph of s_{foo}. However, the actual parameter `c2` is locked in s_{bar1}, so the corresponding formal parameter `c1` is removed from the graph of s_{foo}. The last node in `foo()` corresponds to `lock`, which is a field reference rather than a formal parameter; thus, its program point is replaced with $pp_\perp$ before integrating into `bar()`. The result, s_{bar2}, has one new node ($pp_\perp$) and two new edges (from `c2` to both `b2` and $pp_\perp$). The other state components in s_{bar2} are unchanged from s_{bar1}.

The last component of Figure 10 gives the overall lock-order graph, treating `foo()` and `bar()` as a library of methods. As there is no subclassing in this example, the final lock-order graph can be obtained simply by taking the union of graphs from s_{foo} and s_{bar2}, setting all program points to $pp_\perp$. The cycle in the lock-order graph corresponds to a real deadlock possibility in which `foo()` and `bar()` are called concurrently with the same arguments.



```
1 class A {
2   B lock;
3
4   public
5   void foo(B b1, C c1) {
6     synchronized (b1) {
7       synchronized (c1) {
8         ...
9       }
10    }
11    synchronized (lock) {
12      ...
13    }
14  }
15
16  public
17  synchronized void bar(B b2,
18                        C c2) {
19    synchronized (b2) {
20      ...
21    }
22    synchronized (c2) {
23      foo(b2, c2);
24    }
25  }
26 }
```

s_{foo}
g =
$\langle pp_{b1:5}, B\rangle$
$\langle pp_{lock:11}, B\rangle$
$\langle pp_{c1:5}, C\rangle$
$\text{roots} = \{\langle pp_{b1:5}, B\rangle, \langle pp_{lock:11}, B\rangle\}$
$\text{locks} = \{\}$
$\text{env} = \{\texttt{this} \mapsto \langle pp_{this:5}, A\rangle, \texttt{b1} \mapsto \langle pp_{b1:5}, B\rangle, \texttt{c1} \mapsto \langle pp_{c1:5}, C\rangle\}$
s_{bar1}
g =
$\langle pp_{this:16}, A\rangle$
$\langle pp_{b2:16}, B\rangle$
$\langle pp_{c2:16}, C\rangle$
$\text{roots} = \{\langle pp_{this:16}, A\rangle\}$
$\text{locks} = \{\langle pp_{this:16}, A\rangle, \langle pp_{c2:16}, C\rangle\}$
$\text{env} = \{\texttt{this} \mapsto \langle pp_{this:16}, A\rangle, \texttt{b2} \mapsto \langle pp_{b2:16}, B\rangle, \texttt{c2} \mapsto \langle pp_{c2:16}, C\rangle\}$
s_{bar2}
g =
$\langle pp_{this:16}, A\rangle$
$\langle pp_{b2:16}, B\rangle$
$\langle pp_{c2:16}, C\rangle$
$\langle pp_\perp, B\rangle$
$\text{roots} = s_{bar1}.\text{roots}$
$\text{locks} = s_{bar1}.\text{locks}$
$\text{env} = s_{bar1}.\text{env}$
$\langle pp_\perp, A\rangle$
$\langle pp_\perp, B\rangle$
$\langle pp_\perp, C\rangle$
Overall lock-order graph for foo and bar.

Fig. 10. Example operation of the dataflow analysis. The symbolic state is shown for the method summary of `foo`, as well as for two points in `bar` (before and after a call to `foo`). The `wait` sets (not shown) are empty in each case. The top-level lock-order graph for this library of methods is shown at bottom left

4.4 Reporting Possible Deadlock

To report deadlock possibilities, the analysis finds each cycle in the lock-order graph, using a modified depth-first search algorithm. Once a cycle is found, a report is constructed using its edge annotations. Each edge in the lock-order graph has a pair of annotations, one for the source lock and one for the destination lock. Each annotation consists of the variable name of the lock and the method that acquires it. As graphs are combined, edges may come to have multiple annotations.

A report is given for each distinct set of lock variables. These reports include each of the sets of methods that acquire that set of locks. In this way, methods with the same or similar locking behavior are presented to the user together. In our experience with the tool, most of the grouped method sets constitute the same locking pattern, so this style can save significant user effort.

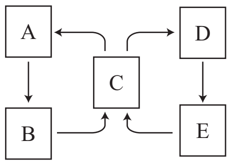


Fig. 11. The path {C, A, B, C, D, E} is a non-simple cycle: it visits node C twice

```
public void println(String s)
{
  synchronized (this) {
    print(s);
    newLine();
  }
}
```

```
public void print(String s) {
  if (s == null) {
    s = "null";
  }
  write(s);
}
```

```
private void write(String s) {
  try {
    synchronized (this) {
      ...
    }
  }
  ...
}
```

Fig. 12. Code excerpt from Sun's `java.io.PrintStream` class. Due to the repeated synchronization on `this`, an intraprocedural analysis reports a spurious deadlock possibility while an interprocedural analysis does not

The analysis reports every *simple* cycle (also known as an elementary circuit) in a given graph. A cycle is simple if it does not visit any node more than once. Given a node that is involved in more than one simple cycle, one can construct a non-simple cycle by traversing each cycle in sequence (see Figure 11). It is possible to construct cases where a non-simple cycle causes deadlock even though the component cycles do not [26]. However, as we have never observed such a case in practice, the analysis reports only the simple cycles as a way of compressing the results. For completeness, the user should consider these cycles in combination.

4.5 Intraprocedural Weaknesses

Our analysis is interprocedural, because our experience is that an intraprocedural analysis produces too many false reports. For example, Figure 12 illustrates part of Sun's `java.io.PrintStream` class, in which both `println()` and `write()` attempt to lock `this`. An intraprocedural analysis cannot prove that the same object is locked in both methods. Thus, it reports a deadlock possibility corresponding to the case when two concurrent calls to `println()` result in different locking orders on a pair of `PrintStream` objects. However, because the objects locked are always equivalent, the second synchronization does not affect the set of locks held. This spurious report is omitted by our interprocedural analysis.

5 Reducing False Positives

Like many static analyses, our tool reports false positives. A false positive is a report that cannot possibly lead to deadlock, because (for example) it requires an infeasible aliasing relationship or an infeasible set of paths through the program. False positives reduce the usability of the tool because verifying that the report is spurious can be tedious. We have implemented sound optimizations that reduce the number of false reports without eliminating any true reports. This section describes some of the optimizations; two additional implemented optimizations handle synchronization over an object and one of its internal fields, and synchronization over method call return values [26].

5.1 Unaliased Fields

An unaliased field is one that always points to an object that is not pointed to by another variable in the program. As an optimization, our analysis detects these fields, and assigns a unique type to each of them. This can decrease the number of deadlock reports by disambiguating unaliased field references from other nodes in the lock-order graph. (It is necessary to create a node for these fields, rather than discarding information about synchronization over them. Although they have no aliases, they may still be involved in deadlock.)

The following analysis is used to discover unaliased fields. Initially, all non-public fields are assumed to be unaliased. As the analysis visits each statement in the library, that assumption is nullified for a field `f` if any of the following patterns apply:

1. `f` is assigned a non-null, non-newly-allocated expression.
2. `f` appears on the right-hand side of an assignment expression.
3. `f` appears outside any of the following expressions: a `synchronized` statement target, a comparison expression (e.g., `foo == bar`), an array access or array length expression, or as an argument to a method that does not allow it to escape.

A simple iterative escape analysis determines which arguments escape a method. Calls from the library to its methods as well as calls to the JDK are checked; arguments are assumed to escape methods where no source is available.

The analysis presented in Section 4.1 introduces a new symbolic heap object for every reference to a field. This is necessary because the analysis does not model the possible values of fields. Unaliased fields are restricted in the possible values they may hold. In particular, they are always assigned new objects, and, if they are reassigned, their old objects cannot be accessed. Because of this property, nested synchronization over the same field of a given object can be treated as a no-op (thereby eliminating spurious reports), since only one of the values locked is accessible. That is, one of the two synchronized statements is on a lock that no longer exists and should therefore be ignored. The analysis uses the same heap object for all references to the same unaliased field within a given object, thereby regarding nested synchronizations as no-ops as desired. This heap object propagates across call sites rather than being mapped to $pp_\perp$.

In addition to detecting unaliased fields, our analysis stores the set of possible runtime types of these fields. This information is readily available for unaliased fields, as they are only assigned fresh objects (created with the `new` keyword).

With this information, the analysis can determine a more precise set of possible callee methods when an unaliased field is used as a receiver.

Detecting and utilizing unaliased fields can be very beneficial. For example, this optimization reduces the number of reports from over 909 to only 1 for the jcurzez library, and from 66 to 0 for the httpunit library.

5.2 Callee/Caller Type Resolution

Accurate knowledge about dynamic types prevents locks on one object from being conservatively assumed to apply to other objects. In general, the dynamic types of arguments are a subclass of the declared parameter types; likewise, the dynamic type of the receiver is a subclass of its declared type in the caller. Callee/caller type resolution collects extra type information by leveraging the fact that the declared types of objects in callees and callers sometimes differ.

To understand the benefits of type resolution, consider the following:

```
Object o;
o.hashCode();
```

When analyzing a particular implementation of `hashCode()`, say, in class `Date`, the receiver is known to be of type `Date`, not `Object` as it was declared in the above code. The callee/caller type resolution optimization takes advantage of this information when integrating the lock-order graph for a callee such as `Date.hashCode()` into that of the caller. Instead of using the callee or caller type exclusively, the more specific type is used. This results in more precise type information in the overall lock-order graph, thereby decreasing the size of the alias sets. Type resolution can have a large impact on spurious reports: reports for the croftsoft library decrease from 1837 to 2, and reports for the jasperreports library decrease from 28 to 0.

5.3 Final and Effectively-Final Fields

For `final` fields, all references are to the same object. Our analysis takes advantage of this fact by using the same heap object for each of the references to the same `final` field within a given object. The analysis also detects fields that are *effectively-final*: non-public fields that are not assigned a value (except null) outside their constructor. Exploiting `final` fields reduces the number of reports from 46 to 32 for the Classpath library.

6 Results

We implemented our deadlock detector in the Kopi Java compiler [9], which inputs Java source code. Our benchmarks consist of 18 libraries, most of which we obtained from SourceForge and Savannah[2]. The results appear in Figure 13.

[2] ProActive [16], Jess [12], SDSU [20], and Sun's JDK [23] are not from SourceForge or Savannah, but are freely available online.

	Code size			Graph size			
Library	`sync`	Classes	kLOC	Nodes	Edges	Reports	Deadlocks
JDK 1.4	1458	1180	419	65	278	70 $*$	≥ 7
Classpath 0.15	754	1074	295	15	22	32 $*$	≥ 5
ProActive 1.0.3	199	407	63	3	3	3 $*$	≥ 2
Jess 6.1p6	111	125	27	12	30	23 $*$	≥ 0
sdsu (1 Oct 2002)	69	139	26	2	2	3 $*$	≥ 0
jcurzez (12 Dec 2001)	24	27	4	1	1	1	0
httpunit 1.5.4	17	117	23	0	0	0	0
jasperreports 0.5.2	11	271	67	0	0	0	0
croftsoft (09 Nov 2003)	11	108	14	1	1	2	0
dom4j 1.4	6	155	41	1	1	1	0
cewolf 0.9.8	6	98	7	0	0	0	0
jfreechart 0.9.17	5	396	125	0	0	0	0
htmlparser 1.4	5	111	22	1	1	0	0
jpcap 0.01.15	4	58	8	0	0	0	0
treemap 2.5.1	4	47	7	0	0	0	0
PDFBox 0.6.5	2	127	28	0	0	0	0
UJAC 0.9.9	1	255	63	0	0	0	0
JOscarLib 0.3beta1	1	77	6	0	0	0	0

$*$ Unsound filtering heuristics used (see Section 6.3)

Fig. 13. Number of deadlock reports for each library. The table indicates the size of each library in terms of number of `synchronized` statements (given in the column labeled `sync`), number of classes (source files), and number of lines of code (in thousands). The size of the lock-order graph is measured after pruning nodes and edges that are not part of a strongly connected component. "Deadlocks" shows the numbers of confirmed deadlock cases in each library. The JDK and Classpath results are for packages in java.*. We were unable to compile 6 source files in JDK due to bugs in our research compiler

The analysis ran in less than 3 minutes per library on a 3.60GHz Pentium 4 machine. For the larger libraries, it is prohibitively expensive to compute all possible deadlock reports, so we implemented a set of unsound heuristics to filter them (see Section 6.3).

6.1 Deadlocks Found

We invoked 14 deadlocks in 3 libraries; 12 of these deadlocks were previously unknown to us. We verified each instance by writing client code that causes deadlock in the library. There are at least 7 deadlocks in the JDK, 5 in GNU Classpath, and 2 in ProActive.

As described in Section 4.4, our analysis groups reports based on the lock variables involved. Some of the deadlocks described below can be induced through calls to any of a number of different methods with the same locking pattern; we only describe a single case, and report the number of deadlocks in this conservative fashion.

Deadlocks Due to Cyclic Data Structures. Of the 14 deadlocks we found, 7 are the result of cycles in the underlying data structures. As an example, consider `java.util.Hashtable`. This class can be deadlocked by creating two `Hashtable` objects and adding each as an element of the other, i.e., by forming a cyclic relationship between the instances. In this circumstance, calling the synchronized `equals()` method on both objects in different threads can yield deadlock. The `equals()` method locks its receiver and calls `equals()` on its members, thus locking any of its internal `Hashtable` objects. When run in two threads, each of the calls to `equals()` has a different lock ordering, so deadlock can result.

Although this example may seem degenerate, the JDK `Hashtable` implementation attempts to support this cyclic structure: the `hashCode()` method prevents a potential infinite loop in such cases by preventing recursive calls from executing the hash value computation. A comment within `hashCode()` says, “This code detects the recursion caused by computing the hash code of a self-referential hash table and prevents the stack overflow that would otherwise result.”

In addition to `Hashtable`, all synchronized `Collections` and combinations of such `Collections` (e.g., a `Vector` in a cyclic relationship with a `Hashtable`) can be deadlocked in a similar fashion. This includes `Collections` produced via calls to `Collections.synchronizedCollection()`, `Collections.synchronizedList()`, `Collections.synchronizedSortedMap()`, etc. For the purposes of reporting, all these cases are counted as a single deadlock in both the JDK and Classpath.

Deadlock resulting from cyclic data structures is quite difficult to correct. Locks must be acquired in a consistent order, or they must be acquired simultaneously. To do either of these things requires knowing which objects will be locked by calling a given method. Determining this information without first locking the container object is problematic since its internals may change during inspection. It appears that the only solution is to use a global lock for synchronizing instances of all `Collection` classes. This solution is undesirable, however, because it prevents multi-threaded uses of different `Collection` objects. Library writers may instead choose to leave these deadlock cases in place, but document their existence and describe how to appropriately use the class.

Not only do these cyclic data structures lead to deadlock, but they may also result in a stack overflow due to infinite recursion. A number of the classes having this kind of deadlock also have methods that produce unbounded recursion for the case of cyclic data structures. It seems that these deadlock cases reveal intended structural invariants (i.e., that a parent object is not reachable through its children) about the classes they involve.

The remaining 5 cyclic deadlocks are similar to that described above. Deadlock can be induced in `java.awt.EventQueue` from both JDK and Classpath, in `java.awt.Menu` from JDK, in `java.util.logging.Logger` from Classpath, and in `AbstractDataObject` from Proactive. Each class has a method that allows a cyclic relationship to be formed, and another method (or set of methods) that locks the containing object and the internal one.

Other Deadlock Cases. In addition to the cyclic case described above, ProActive exhibits a subtle deadlock in the `ProxyForGroup` class. Through a sequence of calls, the `asynchronousCallOnGroup()` method of `ProxyForGroup` can be made to lock both `this` and any other `ProxyForGroup`. Instantiating two or more `ProxyForGroup` objects and forcing each to lock the other induces deadlock. The state necessary to produce this scenario is relatively complex. The offending method contains, within four nested levels of control flow, a method call that returns an `Object`; under certain circumstances, the object returned is a `ProxyForGroup`, as needed to produce deadlock. We would not expect a library writer to notice this deadlock possibility without using a tool like ours.

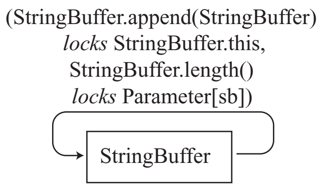
We invoked 4 additional deadlocks in the JDK. One deadlock is in `BeanContextSupport` as described in Section 1. A second deadlock is in `StringBuffer.append(StringBuffer)`, as illustrated in Figure 14. This deadlock occurs because `append()` is a synchronized method (i.e., it locks `this`), and it locks its argument. Thus, using the client code in Figure 14, if `a` is locked in thread 1, and `b` is locked in thread 2 before it is in thread 1, deadlock results. Note that this is an example of a case where only a single method is used to cause deadlock.

Another deadlock from the JDK occurs in `java.io.PrintWriter` and `java.io.CharArrayWriter`. Simplified code for this deadlock is shown in Figure 15. The `PrintWriter` and `CharArrayWriter` classes both contain a `lock` field for synchronizing I/O operations. In `PrintWriter`, the lock is set to the output stream `out`, while in `CharArrayWriter`, the lock is set to `this`.

The last deadlock in the JDK is located in `java.awt.dnd.DropTarget`. This class can be deadlocked by calling `setComponent()` with an argument (of type `Component`) having a valid `DropTarget` set. When this call is made, the receiver is locked followed by the argument's `DropTarget`. Thus, the code in Figure 16 can lead to deadlock.

GNU Classpath exhibits 2 deadlocks besides those described so far. The first is in `StringBuffer`, and is analogous to the JDK bug described above. The second is in `java.util.SimpleTimeZone`. The `SimpleTimeZone.equals(Object)` method is synchronized and locks its argument; it is therefore susceptible to the same style of deadlock as that of `StringBuffer.append()`.

```
class StringBuffer {
 synchronized StringBuffer
   append(StringBuffer sb) {
     ...
     // length() is synchronized
     int len = sb.length();
     ...
 }
}
```


(StringBuffer.append(StringBuffer)
locks StringBuffer.this,
StringBuffer.length()
locks Parameter[sb])
StringBuffer

```
StringBuffer a =
  new StringBuffer();
StringBuffer b =
  new StringBuffer();

thread 1:  a.append(b);
thread 2:  b.append(a);
```

Fig. 14. Library code, lock-order graph, and client code that deadlocks JDK's `StringBuffer` class. This deadlock is also present in Classpath

```
class PrintWriter {
  PrintWriter(OutputStream o) {
    lock = o;
    out = o;
  }

  void write(char buf[],
      int off, int len) {
    synchronized (lock) {
      out.write(buf, off, len);
    }
  }
}
```

```
class CharArrayWriter {
  CharArrayWriter() {
    lock = this;
  }

  void writeTo(Writer out) {
    synchronized (lock) {
      out.write(buf, 0,
                count);
    }
  }
}
```

```
// c.lock = c
c = new CharArrayWriter();
// p1.lock = c
p1 = new PrintWriter(c);
// p2.lock = p1
p2 = new PrintWriter(p1);

thread 1:  p2.write("x",0,1);
thread 2:  c.writeTo(p2);
```

Fig. 15. Simplified library code from `PrintWriter` and `CharArrayWriter` from Sun's JDK, and, on the right, client code that causes deadlock in the methods. In thread 1, `p1` is locked first, then `c`; in thread 2, `c` is locked, then `p1`. Because the locks are acquired in different orders, deadlock occurs under some thread interleavings

```
DropTarget a = new DropTarget(), b = new DropTarget();
Component aComp = new Button(), bComp = new Button();

aComp.setDropTarget(a);
bComp.setDropTarget(b);

thread 1:  a.setComponent(bComp);
thread 2:  b.setComponent(aComp);
```

Fig. 16. Client code that induces deadlock in the JDK's `DropTarget` class

It is interesting to note that JDK and Classpath implementations of `SimpleTimeZone` and `Logger` differ in their locking behavior: it is not possible to invoke deadlock in these classes using the JDK. Similarly, the Classpath implementations of `PrintWriter` and `CharArrayWriter` do not deadlock; other relevant portions of Classpath are not fully implemented.

Fixing Deadlocks. There are a number of viable solutions to the deadlocks presented above. The methods performing synchronization could be written to acquire the needed locks in a set order. Java could be extended with a synchronization primitive to atomically acquire multiple locks. A utility routine could be written to accomplish the same effect as this primitive, taking as arguments a list of locks to acquire and a thunk to execute, then acquiring the locks in a fixed order. These solutions require knowledge of the set of locks to be acquired. Sometimes this is immediately apparent from the code; otherwise, a method that determines the locks required for an operation could be added to an interface. In all these cases, the implementation could order the locks using `System.identityHashCode()`, breaking ties arbitrarily but consistently. Note however, that these solutions assume that the needed locks will not change while they are being determined. If they might change, it may be necessary to use a global lock for the classes involved in the deadlock.

6.2 Verifying Deadlock Freedom

Using our tool, we verified 13 libraries to be free from the class of deadlocks described in Section 3.2. Note that these libraries may perform callbacks to client code, some extend the JDK, and most perform reflection; our technique does not model synchronization resulting from these behaviors. For 10 of these libraries, the verification is fully automatic, with 0 reports from our tool. Across the other 3 libraries, our tool reports a total of 4 deadlocks, which we manually verified to be false positives.

The false report in jcurzez is for a scenario in which an internal field `f` of the same type as its containing class is set to a parameter of the constructor. To eliminate this report, the analysis would have to combine several facts and additional optimizations. Croftsoft gives two spurious reports because an object involved in the synchronization cannot have the runtime type that our tool conservatively assumes to be possible. The final report is for dom4j, and is spurious because of infeasible control flow.

6.3 Unsound Filtering Heuristics

For the larger libraries, the number of reports given by our algorithm is too high (more than 100,000 for the JDK) for each to be considered by hand. In addition, it is computationally demanding to report every deadlock possibility. In order to make the tool more usable for large libraries (both in terms of number of reports and time needed to gather them) our tool uses unsound filtering heuristics. These heuristics aim to identify reports that have the greatest likelihood of representing a true deadlock. However, as unsound heuristics, they also have the potential to eliminate true deadlock cases from consideration.

Our tool applies two filtering heuristics on certain of the libraries in Figure 13. One heuristic is to restrict attention to cycles in the lock-order graph that are shorter than a given length. For the filtered libraries, only cycles with two or fewer nodes were reported. Shorter cycles contain fewer locks, and are easier to examine manually. In addition, shorter cycles might be more likely to correspond to actual deadlocks, as each edge in a cycle represents a pair of lock acquisitions that has some chance of being infeasible (due to infeasible control flow or aliasing relationships).

The second filtering heuristic is to assume that the runtime type of each object is the same as its declared type. This reduces the number of reports in two ways. First, the analysis ceases to account for dynamic dispatch, as it assumes that there is exactly one target of each method call. This causes the lock-order graph for a given method to be integrated at fewer call-sites, thereby decreasing the number of edges in the overall graph. Second, this heuristic causes the **top_level** routine (Figure 8) to forgo expansion of each edge into edges between all possible subclasses. This heuristic has some intuitive merit because it restricts attention to code that operates on a specific type, rather than a more general type. For example, it considers the effects of all synchronized methods of a given class, but it eliminates the assumption that all objects could be aliased with a field of type `Object` that may be locked elsewhere.

7 Related Work

The long-standing goal of ensuring that concurrent programs are free of deadlock remains an active research focus. Mukesh reviews the various approaches [22].

Several researchers have developed static deadlock detection tools for Java using lock-order graphs [17, 1, 24]. To the best of our knowledge, the Jlint static checker [17] is the first to use a lock-order graph. The original implementation of Jlint considers only `synchronized` methods; it does not model `synchronized` statements. Artho and Biere [1] augment Jlint with limited support for `synchronized` statements. However, their analysis does not report all deadlock possibilities. It only considers cases they reason are most fruitful for finding bugs: 1) all fields and local variables are assumed to be unaliased, meaning that two threads must lock exactly the same variable to elicit a deadlock report, 2) nested `synchronized` blocks are tracked only within a single class, not across methods in different classes, and 3) inheritance is not fully considered.

von Praun detects deadlock possibilities in Java programs using a lock-order graph and context-sensitive lock sets [24–pp.105–110]. Our analysis was developed independently [25]. While von Praun's alias analysis is more sophisticated than ours, it is unclear how to adapt it to model all possible calls to a library. Also, in an effort to reduce false positives, the analysis suppresses reports in which all locks belong to the same alias set; as a consequence, it does not find 12 of the 14 deadlocks exposed by our tool. While von Praun's analysis could be trivially modified to report such cases, it would then report, in addition, all of the benign cases that repeatedly lock a single object (as in Figure 12). Suppressing these reports is the motivation for the flow-sensitive and interprocedural aspects of our analysis: our analysis can recognize that two object references are identical, thereby qualifying repeated synchronizations on a given object as benign. von Praun's analysis does not offer this benefit, in part because it is flow-insensitive and unification-based. Also, it does not consider that `wait()` can introduce a cyclic locking pattern (as in Figure 9). Our tool reports all deadlock possibilities.

RacerX [10] is a flow-sensitive, context-sensitive tool for detecting deadlocks and race conditions in C systems code. Because our tool analyzes Java instead of C, it operates under a different set of constraints. We fully account for objects and inheritance, reporting all deadlock possibilities; RacerX operates on a procedural language, and might fail to report every deadlock case due to function pointers and high-overhead functions. Our tool analyzes unmodified Java code, while RacerX requires annotations to indicate the locking behavior of system-specific C functions. Our tool exploits the hierarchical synchronization primitives in Java; in C, precision is sacrificed due to the decoupling of lock and unlock operations (sometimes on different paths of the same function, as noted by the authors).

Several groups have taken a model-checking approach to finding deadlock in Java programs. Demartini, Iosif, and Sisto [8] translate Java into the Promela language, for which the SPIN model checker verifies deadlock freedom. Their verification reports all deadlock possibilities so long as the program does not exceed the maximum number of modeled objects or threads.

Java Pathfinder also performs model checking by translating Java to Promela, including support for exceptions and polymorphism [13]. It has also been used to analyze execution traces; a deadlock vulnerability is reported if two threads obtain locks in a different order at runtime [14]. This approach can detect "gate locks": a shared lock that guards each thread's entry into a hazardous out-of-order locking sequence, thereby preventing deadlock. The technique has evolved into a general online monitoring environment called Java PathExplorer [15].

Breuer and Valls describe static detection of deadlock in the Linux kernel [3]. They target deadlocks caused by threads that call `sleep` while still holding a spinlock. Chaki et al. [4] use counterexample-guided abstraction refinement and the MAGIC verification tool [5] to detect deadlock in message-passing C programs. The technique is compositional and efficient (compared to traditional model checking) because the abstraction for each thread can be refined independently until the overall system exhibits a bug or is proven free of deadlock. However, the number of threads and locks (and their interaction) must be known statically.

The Ada programming language allows rendezvous communication between a call statement in one task and an accept statement in another. Most analyses for Ada aim to verify that rendezvous communication succeeds, rather than considering the order of synchronization on shared resources (locks). For example, Masticola and Ryder [19] give a polynomial-time algorithm for reporting all possible rendezvous deadlocks for a subset of Ada (they also report false positives). Corbett [7] evaluates three methods for finding deadlock in Ada programs. Many analyses rely on the common case where Ada tasks are fixed and initiated together, in contrast to Java threads which are always created dynamically.

Boyapati, Lee, and Rinard [2] augment Java with ownership types to ensure deadlock freedom at compile time. While this is an elegant solution, it requires translating existing programs to use new type annotations, and some computations might be hard to express. Flanagan and Qadeer describe a type and effect system for atomicity [11]. In this system, a method is atomic if it appears to execute serially, without interleaving of other threads. They identify an atomicity violation in `StringBuffer.append`, providing part of the impetus for our work.

Zeng and Martin augment a Java Virtual Machine with a deadlock avoidance mechanism [28]. This technique constructs a lock-order graph dynamically, tracking the actual objects that are locked during execution. As cycles form in the graph, "ghost locks" are introduced to prevent multiple threads from entering the cyclic regions. While this avoids deadlock later in the execution, deadlock could still occur while the graph is being built.

Zeng describes a system that uses exceptions to indicate various kinds of deadlock in a Java Virtual Machine [27]. Such a mechanism allows a client to intelligently respond to deadlock in a library component. Pulse [18] is an operating system mechanism that detects general deadlocks via speculative execution of blocked processes. There is also a large body of work on dynamically detecting deadlock in the context of databases and distributed systems [21, 22].

8 Conclusions

Library writers wish to ensure their libraries are free of deadlock. Because this assurance is difficult to obtain by testing or by hand, a tool for identifying possible deadlock (or verifying freedom from deadlock) is desirable. Model checking is a possible approach to the problem, but the well-known state explosion problem makes it impractical for most libraries.

We have presented a flow-sensitive, context-sensitive analysis for static detection of deadlock in Java libraries. Out of 18 libraries, we verified 13 to be free of deadlock, and found 14 reproducible deadlocks in 3 libraries. The analysis uses lock-order graphs to represent locking configurations extracted from libraries. Nodes in these graphs represent alias sets, edges represent possible lock orderings, and cycles indicate possible deadlocks.

Our analysis is quite effective at verifying deadlock freedom and finding deadlock, but it still produces a sizable number of false reports. Rather than asking the user to investigate these reports, the reports could be dispatched to a model checker which could automatically check for deadlock. In this framework, our tool would serve to limit the search space of the model checker, possibly allowing sound verification of large libraries.

Just as static verification of all possible program executions offers stronger guarantees than dynamic analysis of one or a few executions, verification that a library cannot deadlock is preferable to checking that a particular client program does not deadlock while using the library. To our knowledge, our tool is the first to address the problem of deadlock detection in libraries. However, the technique is also applicable to whole programs, and may prove to be effective in that context.

Acknowledgments

We thank Viktor Kuncak, Manu Sridharan, Huu Hai Nguyen, Wilson Hsieh, and Stephen McCamant for their feedback and suggestions on this work, and Martin Lackner for support with Kopi. The second author thanks Saman Amarasinghe for supporting his participation in this project. We also thank the anonymous reviewers for their comments. This work is supported in part by NSF grant CCR-0133580, the MIT-Oxygen Project, and an NSF Graduate Research Fellowship.

References

1. Artho, C., Biere, A.: Applying static analysis to large-scale, multi-threaded Java programs. In: ASWEC. (2001) 68–75
2. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: preventing data races and deadlocks. In: OOPSLA. (2002) 211–230
3. Breuer, P.T., Garcia-Valls, M.: Static deadlock detection in the Linux kernel. In: Ada-Europe. (2004) 52–64
4. Chaki, S., Clarke, E., Ouaknine, J., Sharygina, N.: Automated, compositional and iterative deadlock detection. In: MEMOCODE. (2004)
5. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. IEEE TSE **30** (2004) 388–402
6. Chase, D.R., Wegman, M., Zadeck, F.K.: Analysis of pointers and structures. In: PLDI. (1990)
7. Corbett, J.C.: Evaluating deadlock detection methods for concurrent software. IEEE TSE **22** (1996) 161–180
8. Demartini, C., Iosif, R., Sisto, R.: A deadlock detection tool for concurrent Java programs. Software: Practice and Experience **29** (1999) 577–603
9. DMS Decision Management Systems GmbH: The Kopi Project (2004) `http://www.dms.at/kopi/`.
10. Engler, D., Ashcraft, K.: RacerX: Effective, static detection of race conditions and deadlocks. In: SOSP. (2003) 237–252
11. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: POPL. (2003) 338–349
12. Friedman-Hill, E.: Jess, the Java expert system shell (2004) `http://herzberg.ca.sandia.gov/jess/`.
13. Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder. STTT **2** (2000) 366–381
14. Havelund, K.: Using runtime analysis to guide model checking of Java programs. In: SPIN. (2000) 245–264
15. Havelund, K., Roşu, G.: Monitoring Java programs with Java PathExplorer. In: RV. (2001)
16. INRIA: Proactive (2004) `http://www-sop.inria.fr/oasis/ProActive/`.
17. Knizhnik, K., Artho, C.: Jlint (2005) `http://jlint.sourceforge.net/`.
18. Li, T., Ellis, C.S., Lebeck, A.R., Sorin, D.J.: Pulse: A dynamic deadlock detection mechanism using speculative execution. In: USENIX Technical Conference. (2005) 31–44
19. Masticola, S.P., Ryder, B.G.: A model of Ada programs for static deadlock detection in polynomial time. Workshop on Parallel and Distributed Debugging (1991)
20. San Diego State University: SDSU Java library (2004) `http://www.eli.sdsu.edu/java-SDSU/`.
21. Shih, C.S., Stankovic, J.A.: Survey of deadlock detection in distributed concurrent programming environments and its application to real-time systems. Technical report, UMass UM-CS-1990-069 (1990)
22. Singhal, M.: Deadlock detection in distributed systems. IEEE Computer **22** (1989) 37–48
23. Sun Microsystems, Inc.: Java Development Kit (2004) `http://java.sun.com/`.
24. von Praun, C.: Detecting Synchronization Defects in Multi-Threaded Object-Oriented Programs. PhD thesis, Swiss Federal Institute of Technology, Zurich (2004)

25. Williams, A., Thies, W., Ernst, M.D.: Static deadlock detection in Java libraries. Research Abstract #102, MIT Computer Science and Artificial Intelligence Laboratory (February, 2004)
26. Williams, A.L.: Static detection of deadlock for Java libraries. Master's thesis, MIT Dept. of EECS (2005)
27. Zeng, F.: Deadlock resolution via exceptions for dependable Java applications. In: DSN. (2003) 731–740
28. Zeng, F., Martin, R.P.: Ghost locks: Deadlock prevention for Java. In: MASPLAS. (2004)

Author Index